

目 录

前言

第 1 章 C#简介	1
1.1 开始 (Start)	1
1.2 类型 (Types)	2
1.2.1 预定义类型 (Predefined type)	4
1.2.2 转换 (Conversion)	6
1.2.3 数组类型 (Array)	7
1.2.4 类型系统的统一 (Type System Unification)	9
1.3 变量与参数 (Variables And Parameters)	10
1.4 自动内存管理 (Automatic Memory Management)	14
1.5 表达式 (Expressions)	17
1.6 语句 (Statements)	17
1.7 类 (Class)	20
1.7.1 常量 (Constants)	22
1.7.2 域 (Fields)	23
1.7.3 方法 (Methods)	24
1.7.4 属性 (Properties)	26
1.7.5 事件 (Event)	26
1.7.6 操作符 (Operators)	28
1.7.7 索引 (Index)	29
1.7.8 实例构造函数 (Instance Constructors)	31
1.7.9 析构函数 (Destructors)	32
1.7.10 静态构造函数 (Static Constructors)	32
1.7.11 继承 (Inheritance)	33
1.8 结构 (Structs)	34
1.9 接口 (Interfaces)	35
1.10 委托 (Delegates)	37
1.11 枚举 (Enums)	38
1.12 名字空间与汇编 (Namespaces And Assemblies)	39
1.13 版本 (Versioning)	41
1.14 属性 (Attributes)	44
第 2 章 词法结构	46
2.1 翻译阶段 (Phases of Translation)	46
2.2 语法符号 (Grammar Notation)	46
2.3 词法分析 (Lexical Analysis)	47

2.3.1 输入 (Input)	47
2.3.2 字符输入 (Input Characters)	47
2.3.3 线终端函数 (Line Terminators)	48
2.3.4 注释 (Comments)	48
2.3.5 空格 (White Space)	49
2.4 标识符 (Tokens)	49
2.4.1 单码字符转义序列 (Unicode Character Escape Sequences)	50
2.4.2 标识符 (Identifiers)	51
2.4.3 关键字 (Keywords)	52
2.4.4 字母 (Literals)	53
2.4.5 操作符与标点符号 (Operators And Punctuators)	58
2.5 预处理指令 (Pre-processing Directive)	58
2.5.1 预处理标识符 (Pre-processing Identifiers)	59
2.5.2 预处理表达式 (Preprocessing expressions)	59
2.5.3 预处理声明 (Pre-processing Declarations)	60
2.5.4 #if, #elif, #else, #endif	61
2.5.5 #error 和 #warning	64
2.5.6 #region 和 #endregion	65
2.5.7 #line	65
第3章 基本概念	66
3.1 程序开始 (Program Startup)	66
3.2 程序结束 (Program Termination)	66
3.3 声明 (Declarations)	67
3.4 元素 (Members)	69
3.4.1 名字空间元素 (Namespace Members)	69
3.4.2 结构成员 (Struct Members)	70
3.4.3 枚举成员 (Enumeration Members)	70
3.4.4 类项 (Class Member)	70
3.4.5 接口成员 (Interface Members)	71
3.4.6 数组项 (Array Members)	71
3.4.7 委托成员 (Delegate Members)	71
3.5 成员访问 (Member Access)	71
3.5.1 声明的可访问性 (Declared Accessibility)	71
3.5.2 可访问域 (Accessibility Domains)	72
3.5.3 访问保护 (Protected Access)	74
3.5.4 访问约束 (Accessibility Constraints)	75
3.6 签名和重载 (Signatures And Overloading)	76
3.7 范围 (Scopes)	77
3.7.1 名字的范围 (Name Scopes)	77

3.7.2 名字隐藏 (Name Hiding)	79
3.8 名字空间和类型名字 (Namespace And Type Names)	82
第4章 类型	84
4.1 值类型 (Value Types)	84
4.1.1 默认构造函数 (Default Constructors)	85
4.1.2 结构类型 (Struct Types)	86
4.1.3 简单类型 (Simple Types)	86
4.1.4 整类型 (Integral Types)	87
4.1.5 浮点数类型 (Floating Point Types)	88
4.1.6 十进制类型 (The Decimal Type)	89
4.1.7 布尔类型 (The Bool Type)	90
4.1.8 枚举类型 (Enumeration Types)	90
4.2 引用类型 (Reference Types)	90
4.2.1 类类型 (Class Types)	91
4.2.2 对象类型 (The Object Type)	91
4.2.3 字符串类型 (The String Type)	91
4.2.4 接口类型 (Interface Types)	92
4.2.5 数组类型 (Array Type)	92
4.2.6 委托类型 (Delegate Type)	92
4.3 封箱和非封箱 (Boxing and Unboxing)	92
4.3.1 封箱转换 (Boxing Conversions)	92
4.3.2 非封箱转换 (Unboxing Conversions)	94
第5章 变量	95
5.1 变量分类 (Variable Categories)	95
5.1.1 静态变量 (Static Variables)	95
5.1.2 实例变量 (Instance Variables)	96
5.1.3 数组成员 (Array Elements)	96
5.1.4 值参数 (Value Parameters)	96
5.1.5 引用参数 (Reference Parameters)	96
5.1.6 输出参数 (Output Parameters)	97
5.1.7 局部变量 (Local Variables)	97
5.2 默认值 (Default Values)	97
5.3 明确赋值 (Definite Assignment)	98
5.3.1 初始赋值变量 (Initially Assigned Variables)	100
5.3.2 初始未赋值变量 (Initially Unassigned Variables)	100
5.4 变量引用 (Variable References)	100
第6章 转换	101

6.1 隐式转换 (Implicit Conversions)	101
6.1.1 一致性转换 (Identity Conversion)	101
6.1.2 隐式数值转换 (Implicit Numeric Conversions)	101
6.1.3 隐式枚举转换 (Implicit Enumeration Conversions)	102
6.1.4 隐式参照转换 (Implicit Reference Conversions)	102
6.1.5 封箱转换 (Boxing Conversions)	102
6.1.6 隐式常量表达式转换 (Implicit Constant Expression Conversions)	102
6.1.7 用户自定义隐式转换 (User-defined Implicit Conversions)	103
6.2 显式转换 (Explicit Conversions)	103
6.2.1 显式数值转换 (Explicit Numeric Conversions)	103
6.2.2 显式枚举转换 (Explicit Enum Conversions)	104
6.2.3 显式引用转换 (Explicit Reference Conversions)	105
6.2.4 非封箱转换 (Unboxing Conversions)	105
6.2.5 用户自定义显式转换 (User-Defined Explicit Conversions)	105
6.3 标准转换 (Standard Conversions)	106
6.3.1 标准隐式转换 (Standard Implicit Conversions)	106
6.3.2 标准显式转换 (Standard Explicit Conversions)	106
6.4 用户自定义转换 (User-Defined Conversions)	106
6.4.1 被允许的用户自定义转换 (Permitted User-Defined Conversions)	106
6.4.2 用户自定义转换求值 (Evaluation of User-Defined Conversions)	106
6.4.3 用户自定义隐式转换 (User-Defined Implicit Conversions)	107
6.4.4 用户自定义显式转换 (User-Defined Explicit Conversions)	108
第7章 表达式	109
7.1 表达式分类 (Expression Classifications)	109
7.1.1 分类 (Classifications)	109
7.1.2 表达式的值 (Values of Expressions)	110
7.2 操作符 (Operators)	110
7.2.1 操作符优先与结合性 (Operator Precedence And Associativity)	110
7.2.2 操作符重载 (Operator Overloading)	111
7.2.3 一元操作符重载分解 (Unary Operator Overload Resolution)	112
7.2.4 二进制操作符重载分解 (Binary Operator Overload Resolution)	112
7.2.5 候选用户自定义操作符 (Candidate User-Defined Operators)	113
7.2.6 数提升 (Numeric Promotions)	113
7.3 成员查找 (Member Lookup)	114
7.4 函数成员 (Function Members)	115
7.4.1 自变量列表 (Argument Lists)	116
7.4.2 重载分解 (Overload Resolution)	119
7.4.3 函数成员引用 (Function Member Invocation)	121
7.5 原始表达式 (Primary Expressions)	122

7.5.1 字母 (Literals)	122
7.5.2 简化名 (Simple Names)	122
7.5.3 括弧表达式 (Parenthesized Expressions)	124
7.5.4 成员访问 (Member Access)	124
7.5.5 引用表达式 (Invocation Expressions)	126
7.5.6 成员访问 (Element Access)	128
7.5.7 This 访问 (This Access)	130
7.5.8 基本访问 (Base Access)	130
7.5.9 后缀增量和减量操作符 (Postfix Increment And Decrement Operators)	131
7.5.10 New 操作符 (New Operator)	132
7.5.11 Typeof 操作符 (Type of Operator)	136
7.5.12 检查的和未检查操作符 (Checked And Unchecked Operators)	137
7.6 一元表达式 (Unary Expression)	139
7.6.1 一元加操作符 (Unary Plus Operator)	140
7.6.2 一元减操作符 (Unary Minus Operator)	140
7.6.3 逻辑非操作符 (Logical Negation Operator)	141
7.6.4 按位求补码操作符 (Bitwise Complement Operator)	141
7.6.5 前缀增量和减量操作符 (Prefix Increment And Decrement Operators)	142
7.6.6 CAST 表达式 (Cast Expressions)	142
7.7 算术运算符 (Arithmetic Operators)	143
7.7.1 乘法运算操作符 (Multiplication Operator)	143
7.7.2 除法运算操作符 (Division operator)	144
7.7.3 求余数操作符 (Remainder Operator)	145
7.7.4 加法操作符 (Addition Operator)	146
7.7.5 减法操作符 (Subtraction Operator)	148
7.8 转换操作符 (Shift Operators)	149
7.9 关系操作符 (Relational Operators)	150
7.9.1 整数比较操作符 (Integer Comparison Operators)	151
7.9.2 浮点数比较操作符 (Floating-Point Comparison Operators)	152
7.9.3 十进制比较操作符 (Decimal Comparison Operators)	153
7.9.4 布尔等操作符 (Boolean Equality Operators)	153
7.9.5 枚举比较操作符 (Enumeration Comparison Operators)	153
7.9.6 引用类型相等操作符 (Reference type Equality Operators)	154
7.9.7 字符串相等操作符 (String Equality Operators)	155
7.9.8 委托相等操作符 (Delegate Equality Operators)	156
7.9.9 is 操作符 (The is Operator)	156
7.9.10 as 操作符 (The as Operator)	157
7.10 逻辑操作符 (Logical Operators)	157
7.10.1 整数逻辑操作符 (Integer Logical Operators)	158

7.10.2 枚举逻辑操作符 (Enumeration Logical Operators)	158
7.10.3 布尔逻辑操作符 (Boolean Logical Operators)	159
7.11 附加条件逻辑操作符 (Conditional Logical Operators)	159
7.11.1 布尔条件逻辑操作符 (Boolean Conditional Logical Operators)	159
7.11.2 自定义条件逻辑操作符 (User-Defined Conditional Logical Operators)	160
7.12 条件操作符 (Conditional Operator)	160
7.13 赋值操作符 (Assignment Operators)	161
7.13.1 简单赋值 (Simple Assignment)	161
7.13.2 复合赋值 (Compound Assignment)	164
7.14 表达式 (Expression)	165
7.15 常量表达式 (Constant Expressions)	165
7.16 布尔表达式 (Boolean Expressions)	166
第 8 章 语句	167
8.1 结束点和可达性 (End Points And Reachability)	167
8.2 块 (Blocks)	169
8.3 空语句 (The Empty Statement)	170
8.4 标号语句 (Labeled Statements)	170
8.5 声明语句 (Declaration Statements)	171
8.5.1 局部变量声明 (Local Variable Declarations)	171
8.5.2 局部常量声明 (Local Constant Declarations)	172
8.6 表达式语句 (Expression Statements)	173
8.7 选择语句 (Selection Statements)	173
8.7.1 if 语句 (The If Statement)	173
8.7.2 switch 语句 (The Switch Statement)	174
8.8 iteration 语句 (Iteration Statements)	178
8.8.1 while 语句 (The While Statement)	178
8.8.2 do 语句 (The Do Statement)	179
8.8.3 for 语句 (The For Statement)	179
8.8.4 for each 语句 (The For Each Statement)	180
8.9 jump 语句 (Jump Statements)	182
8.9.1 break 语句 (The Break Statement)	183
8.9.2 continue 语句 (The Continue Statement)	183
8.9.3 goto 语句 (The Goto Statement)	183
8.9.4 return 语句 (The Return Statement)	184
8.9.5 throw 语句 (The Throw Statement)	185
8.10 try 语句 (The Try Statement)	186
8.11 checked 和 unchecked 语句 (The Checked And Unchecked Statements)	189
8.12 lock 语句 (The Lock Statement)	189
8.13 using 语句 (The Using Statement)	190

第9章 名字空间	192
9.1 编译单元 (Compilation Units)	192
9.2 名字空间声明 (Namespace Declarations)	192
9.3 使用指令 (Using Directives)	194
9.3.1 别名指令使用 (Using Alias Directives)	194
9.3.2 名字空间指令使用 (Using Namespace Directives)	197
9.4 名字空间成员 (Namespace Members)	199
9.5 类型声明 (Type Declarations)	200
第10章 类	201
10.1 类声明 (Class Declarations)	201
10.1.1 类修改函数 (Class Modifiers)	201
10.1.2 类基本说明 (Class Base Specification)	202
10.1.3 类主体 (Class Body)	204
10.2 类成员 (Class Member)	204
10.2.1 继承 (Inheritance)	205
10.2.2 new 修改函数 (The New Modifier)	206
10.2.3 访问修改函数 (Access Modifiers)	206
10.2.4 constituent 类型 (Constituent Types)	206
10.2.5 静态和实例成员 (Static And Instance Members)	206
10.3 常量 (Constants)	207
10.4 域 (Fields)	209
10.4.1 静态和实例域 (Static And Instance Fields)	210
10.4.2 readonly 域 (Readonly Fields)	211
10.4.3 域初始化 (Field Initialization)	212
10.4.4 变量初始化 (Variable Initializers)	213
10.5 方法 (Methods)	214
10.5.1 方法参数 (Method Parameters)	216
10.5.2 静态和实例方法 (Static And Instance Methods)	222
10.5.3 虚拟方法 (Virtual Methods)	222
10.5.4 重载方法 (Override Methods)	224
10.5.5 封装方法 (Sealed Methods)	226
10.5.6 抽象方法 (Abstract Methods)	227
10.5.7 外部方法 (External Methods)	228
10.5.8 方法主体 (Method Body)	229
10.5.9 方法重载 (Method Overloading)	230
10.6 属性 (Properties)	230
10.6.1 静态属性 (Static Properties)	231
10.6.2 访问函数 (Accessors)	231

10.6.3 虚拟、封装、重载和抽象访问函数 (Virtual, Sealed, Override And Abstract Accessors)	237
10.7 事件 (Events)	238
10.7.1 事件访问函数 (Event Accessors)	241
10.7.2 静态事件 (Static Events)	243
10.8 索引 (Indexers)	243
10.9 操作符 (Operators)	246
10.9.1 一元操作符 (Unary Operators)	247
10.9.2 二元操作符 (Binary Operators)	248
10.9.3 转换操作符 (Conversion Operators)	248
10.10 实例构造函数 (Instance Constructors)	249
10.10.1 构造函数初始化 (Constructor Initializers)	250
10.10.2 实例变量初始化函数 (Instance Variable Initializers)	251
10.10.3 构造函数执行 (Constructor Execution)	251
10.10.4 默认构造函数 (Default Constructors)	253
10.10.5 局部构造函数 (Private Constructors)	254
10.10.6 可选的构造函数参数 (Optional Constructor Parameters)	254
10.11 静态构造函数 (Static Constructors)	255
10.12 析构函数 (Destructors)	258
第 11 章 结构	259
11.1 结构声明 (Struct Declarations)	259
11.1.1 结构修改函数 (Struct Modifiers)	259
11.1.2 结构接口 (Struct Interfaces)	260
11.1.3 结构主体 (Struct Body)	260
11.2 结构成员 (Struct Members)	260
11.3 类和结构差异 (Class And Struct Differences)	260
11.3.1 值语义 (Value Semantics)	260
11.3.2 继承 (Inheritance)	261
11.3.3 赋值 (Assignment)	261
11.3.4 默认值 (Default Values)	262
11.3.5 装箱和非装箱 (Boxing And Unboxing)	262
11.3.6 this 的意思 (Meaning Of This)	263
11.3.7 域初始化 (Field Initializers)	263
11.3.8 构造函数 (Constructors)	263
11.3.9 析构函数 (Destructors)	263
11.4 结构实例 (Struct Examples)	263
11.4.1 数据库整数类型 (Database Integer Type)	263
11.4.2 布尔型数据库类型 (Database Boolean Type)	265
第 12 章 数组	268

12.1 数组类型 (Array Types)	268
12.2 数组建立 (Array Creation)	269
12.3 数组成员访问 (Array Element Access)	269
12.4 数组成员 (Array Members)	269
12.5 数组方差 (Array Covariance)	269
12.6 数组初始化函数 (Array Initializers)	270
第 13 章 接口	272
13.1 接口声明 (Interface declarations)	272
13.1.1 接口修改函数 (Interface Modifiers)	272
13.1.2 基本接口 (Base Interfaces)	273
13.1.3 接口主体 (Interface Body)	273
13.2 接口成员 (Interface Members)	274
13.2.1 接口方法 (Interface Methods)	275
13.2.3 接口事件 (Interface Events)	275
13.2.4 接口索引 (Interface Indexers)	275
13.2.5 接口成员访问 (Interface Member Access)	276
13.3 全权接口成员名字 (Fully Qualified Interface Member Names)	278
13.4 接口执行 (Interface Implementations)	278
13.4.1 显式接口成员执行 (Explicit Interface Member Implementations)	279
13.4.2 接口映射 (Interface Mapping)	281
13.4.3 接口执行继承 (Interface Implementation Inheritance)	284
13.4.4 接口再执行 (Interface Re-Implementation)	286
13.4.5 抽象类和接口 (Abstract Classes And Interfaces)	288
第 14 章 枚举	289
14.1 枚举声明 (Enum Declarations)	289
14.2 枚举修改函数 (Enum Modifiers)	290
14.3 枚举成员 (Enum Members)	290
14.4 Enum 值和操作 (Enum Values And Operations)	292
第 15 章 委托	293
15.1 委托声明 (Delegate Declarations)	293
15.2 委托实例 (Delegate Instantiation)	294
15.3 多 cast 委托 (Multi-Cast Delegates)	294
15.4 委托引用 (Delegate Invocation)	294
第 16 章 异常	295
16.1 异常原因 (Causes Of Exceptions)	295
16.2 System.Exception 类 (The System.Exception Class)	295

16.3 怎样处理异常 (How Exceptions Are Handled)	295
16.4 常用的异常类 (Common Exception Classes)	296
第 17 章 属性	297
17.1 类属性 (Attribute Classes)	297
17.1.1 AttributeUsage 属性 (The AttributeUsage Attribute)	297
17.1.2 位置和命名参数 (Positional And Named Parameters)	298
17.1.3 属性参数类型 (Attribute Parameter Types)	299
17.2 属性说明 (Attribute Specification)	299
17.3 属性实例 (Attribute Instances)	302
17.3.1 属性的编译 (Compilation Of An Attribute)	302
17.3.2 属性实例的运行期恢复 (Run-time Retrieval Of An Attribute Instance)	303
17.4 保留属性 (Reserved Attribute)	303
17.4.1 AttributeUsage 属性 (The AttributeUsage Attribute)	303
17.4.2 Conditional 属性 (The Conditional Attribute)	304
17.4.3 Obsolete 属性 (The Obsolete Attribute)	307
附录 A 不安全代码	308
附录 B 互操作性	323
参考文献	336

第1章 C# 简介

C#是在 C 和 C++基础上发展起来的一门简单、现代、对象定位、类型安全的程序语言。C#(读作“C sharp”)植根于 C 和 C++语言家族,且与 C 和 C++程序非常接近,C#旨在把 System 的多产性与 C++尚未完善的功能结合起来。

C#是 Microsoft Visual Studio.NET 的一部分。除 C#外,Visual Studio 还支持 Visual Basic、Visual C++,以及撰稿语言 VBScript 和 Jscript。所有这些语言都可访问 Microsoft.NET 平台,后者包括一个普通的执行工具和一个丰富的类库。Microsoft.NET 定义一个普通语言的子集(CLS)及一种混合语言,这种混合语言确保 CLS 所属的语言和类库之间能够通用地操作,且不留任何痕迹。虽然 C#是一门新的语言,但对于 C#开发者来说,它完全可以访问正时兴的工具如 Visual Basic 和 Visual C++所用的丰富的类库。C#本身没有类库。

本章以下几部分将对这门语言的基本特征加以介绍。以后的章节则对其规则及异常现象进行具体的说明,有时也涉及其运算方式。本章的虽然不十分详细,但却简短清晰。其目的是为了向读者简要介绍这门语言,以便读者能够编写简单的程序及对以后几章的阅读。

1.1 开始(Start)

典型的“hello,world”程序的编写方法如下:

```
using System;
class Hello
{
    static void Main() {
        Console.WriteLine("hello, world");
    }
}
```

C#程序的源代码一般储存在一或多个带有扩展名.CS 的文本文件如 hello.cs 中。使用 Visual Studio 提供的命令行编译函数,这个程序编译时可带有命令行指令:

```
CSC    hello.cs
```

该指令产生一个 hello.exe 的可执行程序。这个程序输出:

```
Hello,world
```

此程序严格的检测过程如下:

- 指令 Using System 引用一个由 Microsoft.NET Frame Work 类库提供的名字空间 System。这个名字空间含有方法 Main 中的类 Console。名字空间提供类库中不同元素的等级含义。指令“using”使得作为名字空间成员的类型的名权使用成为可

能。在程序“hello,world”中使用的 `Console.WriteLine` 就是 `System.Console.WriteLine` 的编写形式。

- 方法 `Main` 是类 `hello` 的成员，它具有 `static` 修改函数。因此，它是类 `hello` 而不是该类实例的一个方法。
- 程序的主要入口——即被调用开始执行程序的方法——总是静态方法 `Main`。
- “hello,world”通过使用一个类库进行输出。这种语言本身没有类库，而是使用一个也被其他语言如 `Visual Basic` 和 `Visual C++` 使用的普通的类库。`C` 和 `C++` 的开发者对于那些在程序“hello,world”中出现的语句很感兴趣。
- 该程序不使用 `Main` 的总方法。总体水平不支持方法和变量：这样的元素总是包含在类型声明过程中（如类和结构的声明过程）。
- 该程序既不使用操作符“`::`”，也不使用“`->`”。“`::`”实际上不是一个操作符，而操作符“`->`”只在一小部分程序中使用。分隔符“`.`”被用在复合名字如：`Console.WriteLine` 中。
- 该程序没有前声明，也没有必要使用前声明，因为声明顺序无关紧要。
- 该程序不用 `#include` 引进程序文本。程序的从属关系是根据符号而不是根据文本确定的。此系统消除了用不同语言所编写的程序之间的障碍。例如，类 `console` 也可用其他语言编写。

1.2 类型 (Types)

`C#`支持两种类型：`Value types` 和 `reference types`。值类型包括简单类型（如 `char`, `int` 和 `float`）、枚举类型和结构类型。引用类型包括类类型、接口类型、委托类型和数组类型。

值类型与引用类型的不同点在于，值类型的变量直接包括它们的数据，而引用类型的变量则把引用储存到对象中。引用类型的两个变量可以引用同一个对象。这样，对一个变量的操作就可能影响另一个变量所引用的对象。值类型的每一个变量都具有它们自己的数据拷贝，因此对一个变量的操作不可能影响另一个变量。下面的例子就表明了这种差异。

```
using System;
class Class1
{
    public int Value = 0;
}
class Test
{
    static void Main() {
        int val1 = 0;
        int val2 = val1;
        val2 = 123;
        Class1 ref1 = new Class1();
        Class1 ref2 = ref1;
```



```

        ref2.Value = 123;
        Console.WriteLine("Values: {0}, {1}", val1, val2);
        Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
    }
}

```

该程序输出:

```

Values: 0, 123
Refs: 123, 123

```

对局部变量 `val1` 的赋值不影响局部变量 `val2`，因为两个局部变量属于同一个值类型（即类型 `int`），而值类型的每一个局部变量都单独储存。在常量中，赋值表达式 `ref2.value=123`；既影响 `ref1` 所引用的对象，也影响 `ref2` 所引用的对象。以下几行则另当别论：

```

Console.WriteLine("Values: {0}, {1}", val1, val2);
Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);

```

因为它们表示 `Console.WriteLine` 的一些字符串的格式化过程，这个过程所需要的自变量数其实是可变的。第一个自变量是一个字符串，它可能含有数个座位如 `{0}` 和 `{1}`。每一个座位表示一系列自变量，如 `{0}` 表示第二个自变量，`{1}` 表示第三个自变量等等。在输出结果被传送到控制台之前，每一个自变量值都被其相应自变量格式化的值取代。

开发者可以通过枚举或结构声明定义新的值类型，也可以通过类、接口以及委托声明定义新的引用类型。

下面的例子：

```

using System;
public enum Color
{
    Red, Blue, Green
}
public struct Point
{
    public int x, y;
}
public interface IBase
{
    void F();
}
public interface IDerived: IBase
{

```

```
void G();  
}  
public class A  
{  
    protected virtual void H() {  
        Console.WriteLine("A.H");  
    }  
}  
public class B: A, IDerived  
{  
    public void F() {  
        Console.WriteLine("B.F, implementation of IDerived.F");  
    }  
    public void G() {  
        Console.WriteLine("B.G, implementation of IDerived.G");  
    }  
    override protected void H() {  
        Console.WriteLine("B.H, override of A.H");  
    }  
}  
public delegate void EmptyDelegate();
```

就列举每种类型声明的一到两个例子。下面几部分将对类型声明进行详细说明。

1.2.1 预定义类型 (Predefined type)

C#提供了一系列预定义类型，它们绝大多数对于 C 和 C++ 的开发者来说都是熟悉的。预定义引用类型是 `object` 和 `string`。类型 `object` 是所以起它类型的最基本类型。类型 `string` 表示单码字符串值；类型 `string` 的值不可改变。

预定义的值类型包括带符号的和不带符号的整数类型、浮点数类型及类型 `bool`, `char` 和 `decimal`。带符号的整类型有 `sbyte`, `short`, `int` 和 `long`；不带符号的整类型有 `byte`, `ushort`, `uint`, `ulong`；浮点数类型是 `float` 和 `double`。类型 `bool` 表示布尔型值：这样的值或者是 `true`，或者是 `false`（只有 `true` 和 `false` 两个）。它的引入使得自供代码的编写更容易，且有助于消除十分普遍的 C++ 代码错误，在这类错误中，开发者在应该使用 “==” 的地方错误地使用了 “=”。在 C# 中，下面的例子是有效的：

```
int i = ...;  
F(i);  
if (i = 0) // Bug: the test should be (i == 0)  
    G();
```

因为表达式 `i=0` 的类型是 `int`，且 `if` 语句需要类型为 `bool` 的一个表达式。类型 `char` 表示单码字符。类型 `char` 的变量表示一单个 16 位的单码字符。

`Decimal` 类型用于计算，这样的计算不支持用浮点数表示时产生的四舍五入错误。这种计算的很普通的例子就是财务运算和税收核算以及货币转换。`Decimal` 类型的有意义位数为 28 位。表 1-1 列出的是预定义类型，并且说明了其字母值的表示方法。

表 1-1 预定义类型

类 型	说 明	例 子
<code>object</code>	所有其他类型的最基本类型	<code>object o = null;</code>
<code>string</code>	字符串类型：一个字符串就是一系列单码字符	<code>string s = "hello";</code>
<code>sbyte</code>	8 位带符号整数类型	<code>sbyte val = 12;</code>
<code>short</code>	16 位带符号整数类型	<code>short val = 12;</code>
<code>int</code>	32 位带符号整数类型	<code>int val = 12;</code>
<code>long</code>	64 位带符号整数类型	<code>long val1 = 12;</code> <code>long val2 = 34L;</code>
<code>byte</code>	8 位不带符号整数类型	<code>byte val1 = 12;</code> <code>byte val2 = 34U;</code>
<code>ushort</code>	16 位不带符号整数类型	<code>ushort val1 = 12;</code> <code>ushort val2 = 34U;</code>
<code>uint</code>	32 位不带符号整数类型	<code>uint val1 = 12;</code> <code>uint val2 = 34U;</code>
<code>ulong</code>	64 位不带符号整数类型	<code>ulong val1 = 12;</code> <code>ulong val2 = 34U;</code> <code>ulong val3 = 56L;</code> <code>ulong val4 = 78UL;</code>
<code>float</code>	单精度浮点数类型	<code>float val = 1.23F;</code>
<code>double</code>	双精度浮点数类型	<code>double val1 = 1.23;</code> <code>double val2 = 4.56D;</code>
<code>bool</code>	布尔类型：布尔值有两个：真或假	<code>bool val1 = true;</code> <code>bool val2 = false;</code>
<code>char</code>	字符类型：一个 <code>char</code> 就是一个单码字符	<code>char val = 'h';</code>
<code>decimal</code>	具有 28 个有意义位的精确的十进制类型	<code>decimal val = 1.23M;</code>

每一个预定义类型都是系统所提供类型的一个编写形式。例如，关键字 `int` 指的就是结构 `system.int32`。作为一种风格，关键字的使用比完整的系统类型名更受欢迎。

预定义值类型如 `int` 只在少数情况下被特殊对待，在绝大多数情况下同其他结构一样。操作符重载使得开发者能够定义新的结构类型，这些结构类型的作用与预定义值类型非常相似。例如，结构 `Digit` 可支持同预定义整类型相同的算术操作，且可定义 `Digit` 和预定义类型之间的转换。

预定义类型用操作符重载它们自己。例如，比较操作符 `==` 和 `!=` 对于不同的预定义类型有不同的语义：

- 类型 `int` 的两个表达式如果表示相同的整型值，则认为它们相等。
- 类型 `object` 的两个表达式如果表示同一个对象或都是 `null`，则认为它们相等。

- 类型 `string` 的两个表达式如果每一个字符位置的字符串实例的长度和字符均相同或都是 `null`，则认为它们相等。

下面的例子：

```
class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

输出：

```
True
False
```

因为第一个比较针对的是类型 `string` 的两个表达式，第二个比较针对的是类型 `object` 的两个表达式。

1.2.2 转换 (Conversion)

预定义类型也有预定义转换。如在预定义类型 `int` 和 `long` 之间的转换。C#有两种不同的转换：隐式转换和显式转换。隐式转换不需要详细检查即可安全被执行。例如，从 `int` 到 `long` 的转换就是一个隐式转换。这种转换总能成功，且绝不会丢失信息。显式转换可被明显地执行，如下例所示：

```
using System;

class Test
{
    static void Main() {
        int intValue = 123;
        long longValue = intValue;
        Console.WriteLine("{0}, {1}", intValue, longValue);
    }
}
```

上例把执行了一个从 `int` 到 `long` 的显式转换。

显式转换执行时带有一个 `cast` 表达式。下面的例子使用了一个从 `long` 到 `int` 的显式转换：

```
using System;
```

```

class Test
{
    static void Main() {
        long longValue = Int64.MaxValue;
        int intValue = (int) longValue;
        Console.WriteLine("(int) {0} = {1}", longValue, intValue);
    }
}

```

它输出:

```
(int) 9223372036854775807 = -1
```

因为有一个溢出, `cast` 表达式既能使用隐式转换又能使用显式转换。

1.2.3 数组类型 (Array)

数组有的是单维的有的是多维的。有的为矩形, 有的是锯齿状。单维数组是最普通的类型, 因此它是一个很好的程序入口。下面的例子:

```

using System;

class Test
{
    static void Main() {
        int[] arr = new int[5];

        for (int i = 0; i < arr.Length; i++)
            arr[i] = i * i;

        for (int i = 0; i < arr.Length; i++)
            Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
    }
}

```

建立 `int` 值的一个单维数组, 初始化该数组元素, 并把它们一一打印出来。该程序输出:

```

arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16

```

前面例子所用的类型 `int []` 是一个数组类型。数组类型的表示方法是在一非数组类型后面加上一个或多个级别说明。下面的例子:

```
class Test
{
    static void Main() {
        int[] a1;    // single-dimensional array of int
        int[,] a2;   // 2-dimensional array of int
        int[,,] a3;  // 3-dimensional array of int

        int[][] j2;  // "jagged" array: array of (array of int)
        int[][][] j3; // array of (array of (array of int))
    }
}
```

表示一系列使用数组类型的局部变量声明，该数组类型的元素类型是 `int`。

数组类型是引用类型，数组变量的声明只为对该数组的引用留出空间。数组实例实际上是由数组初始化函数和数组建立表达式建立的。

下面的例子表示多重数组建立表达式：

```
class Test
{
    static void Main() {
        int[] a1;    // single-dimensional array of int
        int[,] a2;   // 2-dimensional array of int
        int[,,] a3;  // 3-dimensional array of int

        int[][] j2;  // "jagged" array: array of (array of int)
        int[][][] j3; // array of (array of (array of int))
    }
}
```

变量 `a1`, `a2` 和 `a3` 表示 *rectangular arrays*，变量 `jz` 表示一个 *jagged array*。这些术语是根据数组的形状确定的。矩形数组的形状总是矩形的。给定数组每一维的长度，其矩形形状也就确定了。例如，`a3` 的三维数组的长度分别为 10、20 和 30，则很容易就能知道这个数组含有 $10 \times 20 \times 30$ 个元素。

变量 `jz` 表示一个锯齿形数组或“数组的数组”。具体来说，`jz` 表示 `int` 的一个数组的数组或类型 `int[]` 的一个单维数组。每一个 `int[]` 变量都可被单独初始化，这使得该数组呈锯齿状。这个例子，每一个 `int[]` 数组的长度不同。具体说，`jz[0]` 的长度是 3，`jz[1]` 的长度是 6，`jz[2]` 的长度是 9。

一个数组的元素类型和形状——矩形或锯齿状以及其维数——包括在其类型之内。其大小——由其每一维的长度表示——则不包括在其类型的范围之内。这种划分在此语言的语句中很清楚，因为每一维的长度是在数组建立表达式而不是在数组类型中被指定的。例如，下面的声明：

```
int[,,] a3 = new int[10, 20, 30];
```

具有 `int[,]` 的一个数组类型以及 `new int[10,20,30]` 的一个数组建立表达式。对局部变量和域声明来说，允许缩写形式，因为没有必要重新声明数组类型。例如，下面的例子：

```
int[] a1 = new int[] {1, 2, 3};
```

可缩写为：

```
int[] a1 = {1, 2, 3};
```

而程序语言则无任何变化。

使用数组初始化函数如 `{1, 2, 3}` 的上下文，决定正在被初始化的数组的类型。下面的例子表明：同一个数组初始化语句可被几种不同的数组使用。

```
class Test
{
    static void Main() {
        short[] a = {1, 2, 3};
        int[] b = {1, 2, 3};
        long[] c = {1, 2, 3};
    }
}
```

在表达上下文中不可能使用数组初始化函数。

1.2.4 类型系统的统一 (Type System Unification)

C# 提供了一个“统一的类型系统”。所有的类型——包括值类型——都由类型 `object` 源生而来。任何值，甚至“原始”（初级）类型的值都可调用对象方法。下面的例子在整型字母上调用由 `object` 定义的方法 `ToString`。

```
using System;
class Test
{
    static void Main() {
        Console.WriteLine(3.ToString());
    }
}
```

下面的例子更有趣：

```
class Test
{
    static void Main() {
        int i = 123;
        object o = i;    // boxing
    }
}
```

```
        int j = (int) o;    // unboxing
    }
}
```

int 值可被转化为 object，然后，再转回为 int。这个例子既说明了 boxing 又说明了 unboxing。当值类型的一个变量需要被转化为引用类型时，就有一个对象 box 被分配以容纳那个值，该值被拷贝到这个 box 中。unboxing 正好相反。当一个对象箱被掷回到其原来的值类型时，其中的值就被复制出这个箱子，而进入其恰当的存储空间。

这个类型系统统一在对象的帮助下提供值类型，它没有引入不必要的值类型。对于 int 值不必作为对象而使用的程序来说，int 值只是 32 位值。对于 int 值需要作为对象而使用的程序来说，这种可能性在必要时可得到。这种把值类型当作对象来处理的能力，在绝大多数语言中都有的值类型和引用类型之间架起了一座桥梁。例如，一个 stack 类可以提供具有 object 值，且返回 object 值的 Push 和 Pop 方法。

```
public class Stack
{
    public object Pop() {...}

    public void Push(object o) {...}
}
```

因为 C# 具有一个统一的类型系统，stack 类可与任何类型的元素，包括值类型如 int 一起被使用。

1.3 变量与参数 (Variables And Parameters)

变量指的是有存储空间。每一个变量都有一个类型，它决定什么值可被储存在这个变量中。局部变量是在方法、属性或索引中声明的变量。局部变量一般通过指定一个类型名和一个声明函数而被定义，该声明函数指定变量的名字及一个可选择的初始值。如在下例中：

```
int a;
int b = 1;
```

但是，一个局部变量声明也可以含有多个声明函数。a 和 b 的声明可写为：

```
int a,b=1;
```

局部变量在获得其值之前必须被明确赋值。下面的例子是无效的：

```
class Test
{
    static void Main() {
        int a;
        int b = 1;
        int c = a + b;
    }
}
```



```

        ...
    }
}

```

因为它在被赋值之前试图使用变量 `a`。这些规则控制着在 5.3 节中定义的确切赋值。它们的一个 `field` (10.4 节) 就是与一个类、结构或实例有关的变量。声明时带有 `static` 修改函数的域定义一个 `static variable`，声明时没有 `static` 修改函数的域定义一个 `instance variable`。静态域与类型有关，而一个实例变量则与一个实例有关。下面的例子表示一个 `Employee` 类，这个 `Employee` 类具有一个私用静态变量和两个公用实例变量。

```

using System.Data;

class Employee
{
    private static DataSet ds;

    public string Name;
    public decimal Salary;

    ...
}

```

形参声明也定义变量。有四种参数：值参数、引用参数、输出参数和参数数组。“`value parameter`”的作用是“引入”参数传递，在这种传递中，自变量的值被传递给方法，且参数的修改不影响原来的自变量。值参数表示它自己的变量，这与相应自变量不同。这个变量是通过复制相应自变量的值被初始化的。下面的例子表明具有值参数 `p` 的一个方法 `F`：

```

using System;

class Test {
    static void F(int p) {
        Console.WriteLine("p = {0}", p);
        p++;
    }

    F(a);
    Console.WriteLine("post: a = {0}", a);
}
}

```

这个例子输出：

```

pre: a = 1
p = 1
post: a = 1

```

即使值参数 `p` 被修改，结果也一样。

reference parameter 的作用是“引用”参数传递，在这种传递中，参数作为调用者所提供的自变量的一个别名。引用参数本身不定义变量，但引用相应自变量的变量。引用参数一经修改就立即直接影响相应的自变量。引用参数声明时带有一个 `ref` 修改参数。下面的例子表明一个具有两个引用参数的方法 `Swap`：

```
using System;

class Test {
    static void Swap(ref int a, ref int b) {
        int t = a;
        a = b;
        b = t;
    }

    static void Main() {
        int x = 1;
        int y = 2;

        Console.WriteLine("pre: x = {0}, y = {1}", x, y);
        Swap(ref x, ref y);
        Console.WriteLine("post: x = {0}, y = {1}", x, y);
    }
}
```

这个程序输出：

```
pre: x = 1, y = 2
post: x = 2, y = 1
```

在对象声明和使用时都必须使用关键字 `ref`。在调用位使用 `ref` 将把注意力引到参数上，这样，阅读代码的开发者就能理解自变量也可以成为调用的结果。

output parameter 与引用参数相似，所不同的是调用者所提供的自变量的初始化值不重要。输出参数声明时带有一个 `out` 修改函数。下面的例子：

```
using System;

class Test {
    static void Divide(int a, int b, out int result, out int remainder) {
        result = a / b;
        remainder = a % b;
    }

    static void Main() {
        for (int i = 1; i < 10; i++)
            for (int j = 1; j < 10; j++) {
                int ans, r;
                Divide(i, j, out ans, out r);
            }
    }
}
```

```

        Console.WriteLine("{0} / {1} = {2}r{3}", i, j, ans, r);
    }
}

```

表示含有两个输出参数的方法 Divide——这两个输出参数有一个是除的结果，另一个是其余数。

对于值、引用和输出参数来说，调用者所提供的自变量与表示它们的参数之间是一一对应的关系。对于 parameter array 来说，则是多对一的关系：一个参数数组可表示多个自变量。也就是说，参数数组是一个长度可变的自变量列表。

参数数组声明时带有一个 params 修改函数。对于一个给定的方法来说，只有一个参数数组，且它总是最后被指定的函数。一个参数数组的类型总是一个二维数组的类型。调用者既可以传递此数组类型的一个自变量，也可以传递此数组类型的元素类型的任意多个自变量。例如，下面的例子：

```

using System;
class Test
{
    static void F(params int[] args) {
        Console.WriteLine("# of arguments: {0}", args.Length);
        for (int i = 0; i < args.Length; i++)
            Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(new int[] {1, 2, 3, 4});
    }
}

```

就是含有自变量 int 的一个变量数的方法 F 及此方法的几个引用过程。它输出：

```

# of arguments: 0
# of arguments: 1
    args[0] = 1
# of arguments: 2
    args[0] = 1
    args[1] = 2
# of arguments: 3
    args[0] = 1
    args[1] = 2

```

```
args[2] = 3
# of arguments: 4
args[0] = 1
args[1] = 2
args[2] = 3
args[3] = 4
```

这个简介中的绝大多数例子都使用类 `console` 的方法 `WriteLine`。此方法的自变量替代，如下例所示：

```
int a = 1, b = 2;
Console.WriteLine("a = {0}, b = {1}", a, b);
```

是由一个参数数组来完成的。方法 `WriteLine` 为一般情况（即只有少数自变量被传递时）提供几个重载方法，它还提供一个使用参数数组的方法。

```
namespace System
{
    public class Console
    {
        public static void WriteLine(string s) {...}
        public static void WriteLine(string s, object a) {...}
        public static void WriteLine(string s, object a, object b) {...}
        ...
        public static void WriteLine(string s, params object[] args) {...}
    }
}
```

1.4 自动内存管理（Automatic Memory Management）

手工内存管理（Manual Memory Management）需要开发者管理内存块的分配和取消分配情况。手工内存管理既耗时又难办。在 C# 中，自动内存管理的提供使得开发者从这项繁重的工作中解脱出来。在大多数情况下，自动内存管理能够提高代码质量及开发者的生产力，且对表达和执行没有任何副作用。下面的例子：

```
using System;
public class Stack
{
    private Node first = null;
    public bool Empty {
        get {
            return (first == null);
        }
    }
}
```

```

    }
}

public object Pop() {
    if (first == null)
        throw new Exception("Can't Pop from an empty Stack.");
    else {
        object temp = first.Value;
        first = first.Next;
        return temp;
    }
}

public void Push(object o) {first = new Node(o, first);
}

class Node
{
    public Node Next;
    public object Value;
    public Node(object value): this(value, null) {}
    public Node(object value, Node next) {
        Next = next;
        Value = value;
    }
}
}

```

是作为实例 Node 的一个相关列表而执行的类 stack。交叉点实例由方法 push 建立，且当不再需要时由无用单元收集程序收集。当一个 node 实例不能被任何代码访问时，就被无用单元收集程序收集。例如，当某一项被从 stack 中取消时，相关的 Node 实例就被无用单元收集程序收集。下面的例子就是一个使用类 stack 的测验程序：

```

class Test
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 0; i < 10; i++) s.Push(i);
        s = null;
    }
}

```

一个 `stack` 在建立和初始化时有 10 个元素，然后被赋值为 `null`。变量 `s` 一旦被赋值为 `null`，`stack` 以及相关的 10 个 `Node` 实例都将被无用单元收集程序收集。无用单元收集函数可以立即将其清空，只是不需要这样做。

基于 C# 的无用单元收集函数通过围绕内存移动对象而作用，但这种移动对于绝大多数 C# 开发者来说是不可见的。对于那些满足于自动内存管理，但有时需要更好的控制或少许额外操作的开发者来说，C# 提供了编写“不安全”代码的能力。这种代码可直接处理指针类型和“不安全”代码，对于开发者和使用者来说实际上是“安全”的。在带有修改函数 `unsafe` 的代码中必须明确表明不安全代码，以免开发者将其错误地使用，编译者和执行工具必须一起工作以确保不安全代码不被混淆为安全代码。这条规则限制了不安全代码在代码可信任情况下的使用。下面的例子：

```
using System;

class Test
{
    unsafe static void WriteLocations(byte[] arr) {
        fixed (byte *p_arr = arr) {
            byte *p_elem = p_arr;
            for (int i = 0; i < arr.Length; i++) {
                byte value = *p_elem;
                string addr = int.Format((int) p_elem, "X");
                Console.WriteLine("arr[{0}] at 0x{1} is {2}", i,
                                addr, value);

                p_elem++;
            }
        }
    }

    static void Main() {
        byte[] arr = new byte[] {1, 2, 3, 4, 5};
        WriteLocations(arr);
    }
}
```

表示一个不安全方法 `writelocations`。它固定一个数组实例并使用指针操作以重述上面的元素。每一个数组元素的索引、值以及位置都被输送到控制台。此程序的一个可能的输出是：

```
arr[0] at 0x8E0360 is 1
arr[1] at 0x8E0361 is 2
arr[2] at 0x8E0362 is 3
arr[3] at 0x8E0363 is 4
arr[4] at 0x8E0364 is 5
```

当然, 确切的内存地址在此程序的不同执行过程中可能不同。

1.5 表达式 (Expressions)

C# 包括一元操作符、二进制操作符和一个三元操作符。表 1-2 概括了这些操作符, 排列方式是优先级从高到低的顺序。

表 1-2 操作符列表

章 节	检 索 词 汇	操 作 符
7.5	Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
7.6	Unary	+ - ! ~ ++x --x (T)x
7.7	Multiplicative	* / %
7.7	Additive	+ -
7.8	Shift	<< >>
7.9	Relational	< > <= >= is
7.9	Equality	= !=
7.10	Logical AND	&
7.10	Logical XOR	^
7.10	Logical OR	
7.11	Conditional AND	&&
7.11	Conditional OR	
7.12	Conditional	?:
7.13	Assignment	= *= /= %= += -= <<= >>= &= ^= =

如果一个表达式含有多个操作符, 则操作符的 precedence 决定每一个操作符的求值顺序。例如, 表达式 $x+y \times z$ 的求值方式与 $x+(y \times z)$ 相同, 因为操作符 $*$ 比 $+$ 具有更高的优先级。

对于处于两个优先级相同的操作符之间的操作数来说, 这两个操作符的 associativity 决定操作符的执行顺序:

- 除赋值操作符外, 所有的二进制操作符都是 left-associative, 即操作是从左到右进行的。例如, $x+y+z$ 的求值方式为 $(x+y)+z$ 。
- 赋值操作符和条件操作符 ($?:$) 是 right-associative, 即操作是从右到左进行的。如 $x=y=z$ 的求值方式为 $x=(y=z)$ 。

优先级和结合性可由括弧来控制。例如, 式子 $x+y \times z$ 先用 z 乘 y , 其结果再加 x , 但 $(x+y) \times z$ 则先执行 $x+y$, 其结果再乘以 z 。

1.6 语句 (Statements)

尽管有一些很明显的添加和修改, 但 C# 的绝大多数语句都来自于 C 和 C++。表 1-3 列举了这些可用语句的种类, 并附有相应的例子。

表 1-3 C#的可用语句

语 句	例 子
Statement lists and block statements	<pre>static void Main() { F(); G(); { H(); I(); } }</pre>
Labeled statements and goto statements	<pre>static void Main(string[] args) { if (args.Length == 0) goto done; Console.WriteLine(args.Length); done: Console.WriteLine("Done"); }</pre>
Local constant declarations	<pre>static void Main() { const float pi = 3.14; const int r = 123; Console.WriteLine(pi * r * r); }</pre>
Local variable declarations	<pre>static void Main() { int a; int b = 2, c = 3; a = 1; Console.WriteLine(a + b + c); }</pre>
Expression statements	<pre>static int F(int a, int b) { return a + b; } static void Main() { F(1, 2); // Expression statement }</pre>
if statements	<pre>static void Main(string[] args) { if (args.Length == 0) Console.WriteLine("No args"); else Console.WriteLine("Args"); }</pre>
switch statements	<pre>static void Main(string[] args) { switch (args.Length) { case 0: Console.WriteLine("No args"); break; case 1: Console.WriteLine("One arg "); break; default: int n = args.Length; Console.WriteLine("{0} args", n); break; } }</pre>
while statements	<pre>static void Main(string[] args) { int i = 0; while (i < args.Length) { Console.WriteLine(args[i]); i++; } }</pre>

(续)

语 句	例 子
do statements	<pre>static void Main() { string s; do { s = Console.ReadLine(); } while (s != "Exit"); }</pre>
for statements	<pre>static void Main(string[] args) { for (int i = 0; i < args.Length; i++) Console.WriteLine(args[i]); }</pre>
foreach statements	<pre>static void Main(string[] args) { foreach (string s in args) Console.WriteLine(s); }</pre>
break statements	<pre>static void Main(string[] args) { int i = 0; while (true) { if (i > args.Length) break; Console.WriteLine(args[i++]); } }</pre>
continue statements	<pre>static void Main(string[] args) { int i = 0; while (true) { Console.WriteLine(args[i++]); if (i > args.Length) continue; break; } }</pre>
return statements	<pre>static int F(int a, int b) { return a + b; } static void Main() { Console.WriteLine(F(1, 2)); return; }</pre>
throw statements and try statements	<pre>static int F(int a, int b) { if (b == 0) throw new Exception("Divide by zero"); return a / b; } static void Main() { try { Console.WriteLine(F(5, 0)); } catch (Exception e) { Console.WriteLine("Error"); } }</pre>
checked and unchecked statements	<pre>static void Main() { int x = 100000, y = 100000; Console.WriteLine(checked(x * y)); Console.WriteLine(checked(x * y)); // Error Console.WriteLine(x * y); // Error }</pre>
lock statements	<pre>static void Main() { A a = ... lock(a) { a.P = a.P + 1; } }</pre>

语 句	例 子
using statements	<pre> using System; class Resource: IDisposable { public void F() { Console.WriteLine("Resource.F"); } public void Dispose() { } ... } static void Main() { using (Resource r = new Resource()) { r.F(); } } </pre>

1.7 类 (Class)

类声明定义新的引用类型。一个类可以从另一个类继承而来，且可以执行接口。

类成员可能包括常量、域、方法、属性、索引、事件、操作符、构造函数、析构函数及嵌套类型声明。每一个成员都有一个相关的可访问性，它决定可访问该成员的程序文的范围。可访问性有五种可能形式，见表 1-4。

表 1-4 C#中类可访问性的形式

形 式	意 义
public	Access not limited
protected	Access limited to the containing class or types derived from the containing class
internal	Access limited to this program
protected internal	Access limited to this program or types derived from the containing class
private	Access limited to the containing type

下面的例子就是含有每种成员的一个类：

```

using System;

class MyClass
{
    public MyClass() {
        Console.WriteLine("Constructor");
    }

    public MyClass(int value) {
        MyField = value;
        Console.WriteLine("Constructor");
    }
}

```

```
~MyClass() {  
    Console.WriteLine("Destructor");  
}  
  
public const int MyConst = 12;  
public int MyField = 34;  
public void MyMethod(){  
    Console.WriteLine("MyClass.MyMethod");  
}  
  
public int MyProperty {  
    get {  
        return MyField;  
    }  
    set {  
        MyField = value;  
    }  
}  
  
public int this[int index] {  
    get {  
        return 0;  
    }  
    set {  
        Console.WriteLine("this[{0}] = {1}", index, value);  
    }  
}  
  
public event EventHandler MyEvent;  
public static MyClass operator+(MyClass a, MyClass b) {  
    return new MyClass(a.MyField + b.MyField);  
}  
  
internal class MyNestedClass  
{  
}
```

下例即这些成员的用法。

```
class Test  
{  
    static void Main() {  
        // Constructor usage  
        MyClass a = new MyClass();  
        MyClass b = new MyClass(123);  
    }  
}
```

```
// Constant usage
Console.WriteLine("MyConst = {0}", MyClass.MyConst);
// Field usage
a.MyField++;
Console.WriteLine("a.MyField = {0}", a.MyField);
// Method usage
a.MyMethod();
// Property usage
a.MyProperty++;
Console.WriteLine("a.MyProperty = {0}", a.MyProperty);
// Indexer usage
a[3] = a[1] = a[2];
Console.WriteLine("a[3] = {0}", a[3]);
// Event usage
a.MyEvent += new EventHandler(MyHandler);
// Overloaded operator usage
MyClass c = a + b;
}

static void MyHandler(object sender, EventArgs e) {
    Console.WriteLine("Test.MyHandler");
}

internal class MyNestedClass
{
}
}
```

1.7.1 常量 (Constants)

一个常量是表示一常量值的类成员。这个常量值可在编辑期被计算。只要不是循环依赖，在同一个程序内允许某些常量依赖于其他常量。这些规则保证常量表达式定义在 7.15 中的常量表达式中。下面的例子就是一个有两个公用常量的类常量：

```
class Constants
{
    public const int A = 1;
    public const int B = A + 1;
}
```

即使常量被认为是静态成员，常量声明既不需要也不允许有 `static` 修改函数。常量可通过类被访问。如下例：

```

class Test
{
    static void Main() {
        Console.WriteLine("{0}, {1}", Constants.A, Constants.B);
    }
}

```

这个程序打印输出 Constants.A 和 Constants.B 的值。

1.7.2 域 (Fields)

一个域就是一个成员，这个成员表示与一个对象或类有关的变量。下面的例子表示具有内部实例域 `redpart`、`greenpart` 和 `bluepart` 的类 `Color`：

```

class Color
{
    internal ushort redPart;
    internal ushort bluePart;
    internal ushort greenPart;
    public Color(ushort red, ushort blue, ushort green) {
        redPart = red;
        bluePart = blue;
        greenPart = green;
    }
    ...
}

```

如下例所示，域也可以是静态的。

```

class Color
{
    public static Color Red = new Color(0xFF, 0, 0);
    public static Color Blue = new Color(0, 0xFF, 0);
    public static Color Green = new Color(0, 0, 0xFF);
    public static Color White = new Color(0xFF, 0xFF, 0xFF);
    ...
}

```

这个例子就表示 Red、Blue、Green 和 White 的静态域。

对于这种情况来说，静态域并不是一个完美的匹配。在它们被使用之前，这些域在某点被初始化，但这种初始化发生之后，也无法阻止委托改变它们。假设这些值不变，这样一个修改也会在使用 `color` 的其他程序中导致无法预料的错误。只读域 (Readonly fields) 的使用可以防止这些问题的发生。对于只读域的赋值只能作为声明的一部分而发生，或者发生在同一类的构造函数内。因此，类 `color` 可以通过对静态域增添只读修改函数而被扩大：

```
class Color
{
    internal ushort redPart;
    internal ushort bluePart;
    internal ushort greenPart;
    public Color(ushort red, ushort blue, ushort green) {
        redPart = red;
        bluePart = blue;
        greenPart = green;
    }
    public static readonly Color Red = new Color(0xFF, 0, 0);
    public static readonly Color Blue = new Color(0, 0xFF, 0);
    public static readonly Color Green = new Color(0, 0, 0xFF);
    public static readonly Color White = new Color(0xFF, 0xFF, 0xFF);
}
```

1.7.3 方法 (Methods)

一个方法就是一个成员，这个成员执行可被一个对象或类完成的计算或功能。方法都有一个形象列表（可能是空的）、一个返回值（或 void），且或者是静态的，或者是非静态的。static methods 通过类被访问。non-static methods，也叫做 instance methods，则通过类的实例被访问。下例表示具有几个静态方法（clone 和 flip）和几个实例方法（push, pop 和 to string）的一个 stack：

```
using System;
public class Stack
{
    public static Stack Clone(Stack s) {...}
    public static Stack Flip(Stack s) {...}
    public object Pop() {...}
    public void Push(object o) {...}
    public override string ToString() {...}
    ...
}
class Test
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 1; i < 10; i++)
            s.Push(i);
        Stack flipped = Stack.Flip(s);
    }
}
```

```
Stack cloned = Stack.Clone(s);  
Console.WriteLine("Original stack: " + s.ToString());  
Console.WriteLine("Flipped stack:" + flipped.ToString());  
Console.WriteLine("Cloned stack: " + cloned.ToString());  
}  
}
```

方法可被重载，即只要它们都具有各自的签名，多个方法可以使用同一个名字。一个方法的签名包括这个方法的名字数量修改函数及其形参的类型。但一个方法的签名不包括返回类型。下面的例子就是一个具有一系列 F 方法的类：

```
class Test  
{  
    static void F() {  
        Console.WriteLine("F()");  
    }  
    static void F(object o) {  
        Console.WriteLine("F(object)");  
    }  
    static void F(int value) {  
        Console.WriteLine("F(int)");  
    }  
    static void F(int a, int b) {  
        Console.WriteLine("F(int, int)");  
    }  
    static void F(int[] values) {  
        Console.WriteLine("F(int[])");  
    }  
    static void Main() {  
        F();  
        F(1);  
        F((object)1);  
        F(1, 2);  
        F(new int[] {1, 2, 3});  
    }  
}
```

这个程序输出：

```
F()  
F(int)  
F(object)
```

```
F(int, int)
F(int[])
```

1.7.4 属性 (Properties)

一个属性就是一个成员，这个成员可访问对象或类的属性。属性包括字符串的长度、字的大小、窗口的标题、顾客的名字等等。属性是域的自然延伸。它们都是带有相关类型的成员，且可访问域和属性的句法也是相同的。然而，与域不同，属性不表示存储地址。但属性具有访问函数，它们指定为读和写属性的值要执行的语句。

属性在定义时具有属性声明。属性声明的第一部分与域声明非常相似。第二部分包括一个 `get` 访问函数和 / 或一个 `set` 访问函数。下面的例子中，类 `button` 定义属性标题：

```
public class Button
{
    private string caption;
    public string Caption {
        get {
            return caption;
        }
        set {
            caption = value;
            Repaint();
        }
    }
}
```

既能读又能写的属性，如属性标题，既包括 `get` 访问函数又包括 `set` 访问函数。当读属性的值时，调用 `get` 访问函数；当写属性的值时，调用 `set` 访问函数。在 `set` 访问函数内属性的新值在隐式参数 `value` 中被给定。

属性的声明比较直接，但属性的值只在使用时才出现，在声明时不出现。属性标题的读写方式与域相同。

```
Button b = new Button();
b.Caption = "ABC";           // set; causes repaint
string s = b.Caption;       // get
b.Caption += "DEF";         // get & set; causes repaint
```

1.7.5 事件 (Event)

一个事件就是一个成员，它使得对象或类能够提供通知。类通过提供事件声明定义一个事件，它与域声明类似，只是增加了一个关键字 `event` 和事件访问函数的一个可选择的装置。这个声明的类型必须是一个委托类型。

下面的例子：


```

public delegate void EventHandler(object sender, System.EventArgs e);
public class Button
{
    public event EventHandler Click;
    public void Reset() {
        Click = null;
    }
}

```

类 `Button` 定义类型 `EventHandler` 的一个事件 `click`。在类 `Button` 中，成员 `click` 与类型 `EventHandler` 的专用域完全一致。然而，在类 `Button` 之外，成员 `click` 只能被用作操作符“+”和“-”的左边。操作符“+”为事件增加一个处理函数，而操作符“-”则为事件减去一个处理函数。下面的例子就是一个类 `Form1`，它给事件 `Button1` 的 `click` 增加一个 `Button1_Click` 作为事件处理函数：

```

using System;
public class Form1
{
    public Form1() {
        // Add Button1_Click as an event handler for Button1's
        Click event
        Button1.Click += new EventHandler(Button1_Click);
    }
    Button Button1 = new Button();
    void Button1_Click(object sender, EventArgs e) {
        Console.WriteLine("Button1 was clicked!");
    }
    public void Disconnect() {
        Button1.Click -= new EventHandler(Button1_Click);
    }
}

```

在方法 `Disconnect` 中，这个事件处理函数被取消。

对于一个简单的事件声明如：

```

public event EventHandler Click;

```

来说，编译器自动提供基于操作符“+”和“-”的执行。

需要更多控制的执行函数可以通过显式提供增加取消访问函数完成上述操作。类 `Button` 可被重新编写以增加和取消访问函数，如下例所示：

```

public class Button
{
    private EventHandler handler;
    public event EventHandler Click {

```

```
        add { handler += value; }
        remove { handler -= value; }
    }
}
```

这种变化对 client 代码没有影响，但允许类 Button 有更大的执行灵活性。例如，click 的事件处理函数不需要用域来表示。

1.7.6 操作符 (Operators)

一个操作符就是一个成员。这个成员定义可被用到类的实例中的表达式操作符的含义。有三种操作符可被定义：一元操作符、二进制操作符和转换操作符。

下面的例子定义一个类型 Digit，这个类型代表十进制数字一从 0 到 9 的整数值。

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }
    public Digit(int value): this((byte) value) {}
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
    public static Digit operator+(Digit a, Digit b) {
        return new Digit(a.value + b.value);
    }
    public static Digit operator-(Digit a, Digit b) {
        return new Digit(a.value - b.value);
    }
    public static bool operator==(Digit a, Digit b) {
        return a.value == b.value;
    }
    public static bool operator!=(Digit a, Digit b) {
        return a.value != b.value;
    }
}
```

```

public override bool Equals(object value) {
    return this == (Digit) value;
}

public override int GetHashCode() {
    return value.GetHashCode();
}

public override string ToString() {
    return value.ToString();
}

}

class Test
{
    static void Main() {
        Digit a = (Digit) 5;
        Digit b = (Digit) 3;
        Digit plus = a + b;
        Digit minus = a - b;
        bool equals = (a == b);
        Console.WriteLine("{0} + {1} = {2}", a, b, plus);
        Console.WriteLine("{0} - {1} = {2}", a, b, minus);
        Console.WriteLine("{0} == {1} = {2}", a, b, equals);
    }
}

```

类型 `Digit` 定义下面的操作符:

- 从 `Digit` 到 `byte` 的隐式转换操作符。
- 从 `byte` 到 `Digit` 的显式转换操作符。
- 增加两个 `Digit` 值, 返回一个 `Digit` 值的附加操作符。
- 从另一个 `Digit` 值中减去一个 `Digit` 值, 且返回一个 `Digit` 值的减操作符。
- 比较两个 `Digit` 值的相等操作符 (`==`) 和不等操作符 (`!=`)。

1.7.7 索引 (Index)

一般索引就是一个成员, 它使得一个对象能够以与数组相同的方式被索引。属性具有与域相似的访问, 索引则具有与数组相似的访问。作为一个例子, 认为类 `Stack` 出现得较早些。这个类可能需要与数组相似的访问, 以便在不执行 `Push` 和 `Pop` 操作符的情况下, 它能够检查或改变堆栈中的项目。`Stack` 作为一个相关的列表被执行, 但它需要提供数组访问的方便。索引声明与属性声明相似, 其主要区别就在于索引没有名字 (在声明中使用的名字是 `this`, 因为 `this` 正在被索引), 且索引含有索引参数。这些索引参数一般在两个方括弧之间。下面的例子就是类 `Stack` 的一个索引函数:

```
using System;

public class Stack
{
    private Node GetNode(int index) {
        Node temp = first;
        while (index > 0) {
            temp = temp.Next;
            index--;
        }
        return temp;
    }

    public object this[int index] {
        get {
            if (!ValidIndex(index))
                throw new Exception("Index out of range.");
            else
                return GetNode(index).Value;
        }
        set {
            if (!ValidIndex(index))
                throw new Exception("Index out of range.");
            else
                GetNode(index).Value = value;
        }
    }
    ...
}

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(2);
        s.Push(3);
        s[0] = 33;    // Changes the top item from 3 to 33
        s[1] = 22;    // Changes the middle item from 2 to 22
        s[2] = 11;    // Changes the bottom item from 1 to 11
    }
}
```

1.7.8 实例构造函数 (Instance Constructors)

一个实例构造函数就是一个成员，这个成员执行初始化类的实例时所需要的操作。下面的例子就是具有两个公用构造函数的类 Point:

```
using System;

class Point
{
    public double x, y;

    public Point() {
        this.x = 0;
        this.y = 0;
    }

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public static double Distance(Point a, Point b) {
        double xdiff = a.x - b.x;
        double ydiff = a.y - b.y;
        return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
    }

    public override string ToString() {
        return string.Format("{0}, {1}", x, y);
    }
}

class Test
{
    static void Main() {
        Point a = new Point();
        Point b = new Point(3, 4);
        double d = Point.Distance(a, b);
        Console.WriteLine("Distance from {0} to {1} is {2}", a, b,
d);
    }
}
```

一个 Point 构造函数没有自变量，另一个则有两个 double 自变量。

如果没有构造函数提供给一个类，那么一个空的没有参数的构造函数就被自动提供。

1.7.9 析构函数 (Destructors)

一个析构函数就是一个成员。它析构一个类的实例。析构函数不能有参数，可访问的修改函数也不能被显式调用。实例的析构函数在无用单元收集期间被自动调用。

下面的例子就是具有析构函数的一个类 point:

```
using System;

class Point
{
    public double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    ~Point() {
        Console.WriteLine("Destructed {0}", this);
    }
    public override string ToString() {
        return string.Format("{0}, {1}", x, y);
    }
}
```

1.7.10 静态构造函数 (Static Constructors)

一个静态构造函数就是初始化一个类的成员。静态构造函数不能带有参数，不能具有可访问的修改函数，也不能被显式调用。当一个类登陆时，其静态构造函数就被自动调用，例如：

```
using System.Data;

class Employee
{
    private static DataSet ds;
    static Employee() {
        ds = new DataSet(...);
    }
    public string Name;
    public decimal Salary;
    ...
}
```

1.7.11 继承 (Inheritance)

类支持单个继承，类型 `object` 是所有类的最基本类。

出现在较早例子中的类都显式从 `object` 继承而来。下面的例子就是一个显式从 `object` 派生的类 A：

```
class A
{
    public void F() { Console.WriteLine("A.F"); }
}
```

下面的例子就是从 A 派生的类 B：

```
class B: A
{
    public void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();

        b.F();           // Inherited from A
        b.G();           // Introduced in B

        A a = b;         // Treat a B as an A
        a.F();

    }
}
```

类 B 继承 A 的 F 方法，而引入它自己的 G 方法。

方法、属性和索引可以是 `virtual`，这意味着它们的执行可在派生类中被覆盖。下面的例子就是一个具有虚拟方法 F 的类 A 以及一个覆盖 F 的类 B：

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() {
        base.F();
        Console.WriteLine("B.F");
    }
}
```

```
    }  
}  
class Test  
{  
    static void Main() {  
        B b = new B();  
        b.F();  
        A a = b;  
        a.F();  
    }  
}
```

B 中的覆盖方法包括一个调用 base.F(), 后者调用 A 中被覆盖的方法。

一个类可以表明它是被完全抽象的, 且由于含有 **abstract** 修改函数, 只是作为其他类的一个基类。这样一个类被称为一个 **abstract class**。抽象类可以指定 **abstract members**——即非抽象派生类必须执行的成员。下面的例子引入抽象类 A 中的一个抽象方法 F:

```
using System;  
abstract class A  
{  
    public abstract F();  
}  
class B: A  
{  
    public override F() { Console.WriteLine("B.F"); }  
}  
class Test  
{  
    static void Main() {  
        B b = new B();  
        B.F();  
        A a = b;  
        a.F();  
    }  
}
```

非抽象类 B 为这个方法提供一个执行。

1.8 结构 (Structs)

类和结构的相似点很多——结构可以执行接口, 且可与类具有相同种类的成员。然而结构不同于类的几个地方都很重要, 结构是值类型而不是引用类型, 且结构不支持继承。结

构的值既不储存在“堆栈”中，也不储存在“程序线”中。仔细的程序员有时能够通过结构的巧妙使用改善其执行。

例如，在一个 `Point` 中，结构而不是类的使用可以造成程序执行的内存分配数量的很大差异。下面的程序建立并初始化 100 个点的一个数组。对于作为一个类而被执行的 `Point` 来说，这个程序例举了 101 个独立的对象，其中一个针对数组，其他的 100 个分别针对 100 个元素。

```
class Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++)
            points[i] = new Point(i, i*i);
    }
}
```

如果 `Point` 作为一个结构被执行，如下例中：

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

那么，这个测验程序只例举了一个对象——即针对数组的对象。实例 `Point` 被定位到数组的内部线中。这种优化过程可能被误用。用结构来代替类也会使程序的运行速度变慢，体积变得臃肿，因为把一个结构实例作为一个值参数来传递时，必须建立这个结构的一个拷贝。详细的数据结构和几何设计无法被取代。

1.9 接口 (Interfaces)

一个接口定义一个契约。执行接口的类或结构必须遵守这个契约。接口可以含有方法、属性、索引和事件。下面的例子：

```
interface IExample
{
    string this[int index] { get; set; }
    event EventHandler E;
    void F(int value);
    string P { get; set; }
}
```

`public delegate void EventHandler(object sender, EventArgs e);` 就是含有一个索引、事件 E、方法 F 以及属性 P 的一个接口。

接口可以有多个继承。例如：

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}
```

`interface IComboBox: ITextBox, IListBox {}` 接口 `IcomboBox` 既是从 `ITextBox` 继承而来，又是从 `IListBox` 继承而来的。

类和结构可以执行多个接口。例如：

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: Control, IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

类 `EditBox` 从类 `Control` 继承而来，且既执行 `Icontrol` 又执行 `IdataBound`。

在前面的例子中，来自接口 `Icontrol` 的方法 `Paint` 和来自于接口 `IdataBound` 的方法 `Bind` 通过使用类 `EditBox` 中的成员而被执行。C# 用另一种方式执行这些方法，在这些方式中，执行类时所用的成员可以不是公用的。接口成员可以通过权名而被执行。例如，如果有方法 `Icontrol.Paint` 和 `IdataBound.Bind`，类 `EditBox` 可以被执行。

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

用这种方法被执行的接口成员称为 **explicit interface members**，因为每一个成员都显式 **designates** 正在被执行的接口成员。显式接口成员只能通过接口被调用。例如，方法 **Paint** 的 **EditBox** 的执行只能通过掷到接口 **IControl** 而被调用。

```
class Test
{
    static void Main() {
        EditBox editbox = new EditBox();
        editbox.Paint();    // error: no such method
        IControl control = editbox;
        control.Paint();    // calls EditBox's Paint
                             implementation
    }
}
```

1.10 委托 (Delegates)

委托与在 C++ 及其他语言中的函数指针类似。与函数指针不同的是，委托是对象定位、类型安全且保险的。委托是从普通基类派生的引用类型：**System.Delegate**。一个委托实例压缩一个方法——即一个可调用的实体。对于实例方法来说，一个可调用的实体包括一个实例和这个实例上的一个方法。对于静态方法来说，一个可调用的实体包括一个类以及这个类上的一个静态方法。

委托不知道也不关心它们引用的对象的类型，这一点很有趣，也很有用。只要方法的签名与委托的一致，任何对象都可以。这使得委托很适合“匿名”引用。这是一项很强的能力。委托从定义到使用可分为三步：声明、示例和调用。委托用委托声明句法来声明。下面的例子声明一个委托 **Simple Delegate**，它没有自变量且返回 **void**。

```
delegate void SimpleDelegate();
```

下面的例子是建立一个实例 **Simple Delegate**，并立即调用它：

```
class Test
{
    static void F() {
        System.Console.WriteLine("Test.F");
    }
}
```

```
static void Main() {  
    SimpleDelegate d = new SimpleDelegate(F);  
    d();  
}  
}
```

没有必要为一个方法示例一个委托，然后通过该委托立即调用它。因为直接调用一个方法会更简单。在使用匿名时，委托就显示出了其优越性。下面的例子：

```
void MultiCall(SimpleDelegate d, int count) {  
    for (int i = 0; i < count; i++)  
        d();  
}
```

方法 `MultiCall` 不知道也不关心 `SimpleDelegate` 目标方法的类型、可访问性以及该方法是否是静态还是非静态的，只要目标方法的签名与 `SimpleDelegate` 一致即可。

1.11 枚举 (Enums)

枚举类型声明为符号常量的一个相关的群体定义一个类型名。枚举在有多个可选项时使用，在这种选择中，运行期的决定是从编译期已经知道的一定数量的选择中作出的。下面的例子就是一个枚举 `color` 和使用这个枚举的一个方法：

```
enum Color  
{  
    Red,  
    Blue,  
    Green  
}  
  
class Shape  
{  
    public void Fill(Color color) {  
        switch(color) {  
            case Color.Red:  
                ...  
                break;  
            case Color.Blue:  
                ...  
                break;  
            case Color.Green:  
                ...  
                break;  
        }  
    }  
}
```

```

        default:
            break;
    }
}
}

```

方法 Fill 的签名表明其形状可被给定的颜色之一填充。

枚举的使用比整型常量更优越——这在没有枚举的语言中很明显——因为枚举的使用使得代码更可读，且可以自己编制。代码的自制性使得开发工具有助于代码编写及其他“设计者”的工作。例如，用 color 代替 int 作为参数类型使得聪明的编译器能够暗示 color 的值。

1.12 名字空间与汇编 (Namespaces And Assemblies)

现在的程序已经能够自立，而不再依靠那些系统提供的类如 System.console。一个程序含有几个不同的片段则更普遍。例如，一个 corporate 应用可能包括几个不同的部分，其中，有的是内部形成的，有的是从独立的软件 vendors 那里买到的。

namespaces 和 assemblies 就是这样由几部分构成的系统。名字空间是一个逻辑组织系统。名字空间既可用作一个程序的“内部”组织系统，又可用作“外部”组织系统——提供暴露给其他程序的程序元素的一种方式。Assemblies 的作用是进行机械装卸。一个汇编就相当于一个类型容器。汇编可能包括类型、执行这些类型的可执行代码以及对其他汇编的引用。

汇编有两个主要类型：应用 (applications) 和库 (libraries)。应用有一个主要入口且通常有一个文件扩展名.exe；库没有主要入口，且通常有一个文件扩展名.dll。

为了说明名字空间和汇编的用法，让我们再回到前面出现的“hello,world”程序，并把它分成两部分：提供信息的一个库和显示它们的控制台应用。

这个库将含有一个类 HelloMessage。下面的例子就是在一个名字空间 Microsoft.Csharp.Introduction 中的类 HelloMessage：

```

// HelloLibrary.cs
namespace Microsoft.CSharp.Introduction
{
    public class HelloMessage
    {
        public string Message {
            get {
                return "hello, world";
            }
        }
    }
}

```

类 Hello Message 具有只读属性 Message。名字空间可以嵌套。下面的例子：

```
namespace Microsoft.CSharp.Introduction
{
}
```

就是嵌套几层的名字空间的一个缩写:

```
namespace Microsoft
{
    namespace CSharp
    {
        namespace Introduction
        {
            [...]
        }
    }
}
```

要编写“hello,world”这个程序,下一步就是编写一个使用类 **Hello Message** 的控制台应用。也可以使用这个类的全名——**Microsoft.Csharp.Introduction.HelloMessage**——但这个名字太长且不普及。使用 **using namespace directive** 更简单,它允许在名字空间内使用所有类型,且不必使用权名。下面的例子:

```
// HelloApp.cs
using Microsoft.CSharp.Introduction;
class HelloApp
{
    static void Main() {
        HelloMessage m = new HelloMessage();
        System.Console.WriteLine(m.Message);
    }
}
```

就是一个表示名字空间 **Microsoft.Csharp.Introduction** 的 **using namespace directive**。**Hello Message** 是 **Microsoft.Csharp.Introduction.HelloMessage** 的缩写形式。

C#也可可以定义和使用别名。**using alias directive** 定义一个类型的别名。当两个库之间发生名字冲突或在一个较大名字空间内只有少数类型被使用时,就可以使用别名来解决上述问题。下面的例子:

```
using MessageSource = Microsoft.CSharp.Introduction.HelloMessage;
```

就是定义类 **Hello Message** 的别名 **Message Source** 的一个 **using alias directive**(使用别名指令)。

我们所编写的代码可被编译到含有类 **Hello Message** 的库和含有类 **Hello App** 的应用内。由于所用的工具以及编译器的不同,这个编译过程的细节可能有所差异。使用 **Visual Studio 7.0** 中的命令行编译器,正确的调用过程是产生类库 **Hello Library.dll** 的

```
csc /target:library HelloLibrary.cs
```

以及产生应用 **Hello App.exe** 的

```
csc /reference:HelloLibrary.dll HelloApp.cs
```

1.13 版本 (Versioning)

版本 (Versioning) 反映了组成部分合理的进化过程。一个组成部分的新版本就是具有一个先前版本的源相容 (source compatible)，前提是当重新编译时，依赖于以前版本的代码能够与新代码一起工作。而如果不需要重新编译，依靠老版本的程序就能与新版本一起工作，那么组成部分的新版本就是 binary compatible。

绝大多数语言不支持二进制相容性，且许多这些相容性都不能使原 compatible 更加方便使用。实际上，有些语言含有错误，这些错误使得在一般情况下，如果没有打破至少某个代码，不可能使一个类得到发展。

例如，假设运载一个类的基类 author 叫做 Base。在第一个版本中，Base 不含有方法 F。一个组成部分从 Base 派生而来，且引入一个 F。这个 Derived 类，与它所依靠的类 Base 一起被释放给 deploy to numerous clients and servers 的客户。

```
// Author A
namespace A
{
    public class Base           // version 1
    {
    }
}

// Author B
namespace B
{
    class Derived: A.Base
    {
        public virtual void F() {
            System.Console.WriteLine("Derived.F");
        }
    }
}
```

这个程序目前来看很好。但现在，开始出现版本错误。Base 的作者制造了一个新的版本，并添加了它自己的方法 F。

```
// Author A
namespace A
{
    public class Base           // version 2
    {
        public virtual void F() {           // added in version 2
            System.Console.WriteLine("Base.F");
        }
    }
}
```

```
    }  
    }  
}
```

Base 的这个新版本与最初的版本应该既是源相容的，又是二进制相容的。（如果只是添加一个方法不可能，那么，一个基类就无法发展。）不幸的是，Base 针对新的 F 使得 Derived 的 F 的含义不明确。难道 Derived 是要覆盖 Base 的 F 吗？这似乎不可能，因为当 Derived 被编译时，Base 还没有 F！但如果 Derived 的 F 不覆盖 Base 的 F，那么它就必须遵守由 Base 指定的契约（这个契约在编写 Derived 时还未被指定）吗？在有些情况下，这是不可能的。例如，Base 的 F 的契约可能要求其覆盖必须总是调用 Base。Derived 的 F 不可能遵守这样一个契约。

C# 的版本需要开发者明确说明他们的目的。在源代码的例子中，代码是清楚的，因为还没有 F。Derived 的 F 的目的是作为一个新的方法而不是一个基本方法的覆盖，因为不存在名字为 F 的基本方法。

```
// Author A  
namespace A  
{  
    public class Base  
    {  
    }  
}  
  
// Author B  
namespace B  
{  
    class Derived: A.Base  
    {  
        public virtual void F() {  
            System.Console.WriteLine("Derived.F");  
        }  
    }  
}
```

如果 Base 增加一个 F 并装在一个新的版本中，Derived 的一个二进制版本的作用仍然是清楚的——Derived 的 F 在语义上是不相干的，且不应该被当作一个覆盖来对待。

然而，当 Derived 被重新编译时，其含义就是不清楚的——Derived 的作者想用其 F 覆盖 Base 的 F，或者隐藏它。由于这个意图不明确，编译器就产生一个警告，并默认为 Derived 的 F 隐藏 Base 的 F。当 Derived 不被重新编译时，这项功能复制语言。所产生的警告使 Derived 的作者十分惊奇：在 Base 中存在方法 F。

如果 Derived 的 F 在语义上与 Base 的 F 无关，那么，Derived 的作者就能表达这种意图——并且有效地关闭警告——这一过程需要使用 F 声明过程中的关键字 new。


```
// Author A
namespace A
{
    public class Base                // version 2
    {
        public virtual void F() { // added in version 2
            System.Console.WriteLine("Base.F");
        }
    }
}

// Author B
namespace B
{
    class Derived: A.Base            // version 2a: new
    {
        new public virtual void F() {
            System.Console.WriteLine("Derived.F");
        }
    }
}
```

另一方面, `Derived` 的作者可能进一步调查, 并决定 `Derived` 的 `F` 应该覆盖 `Base` 的 `F`。这个意图可以通过使用关键字 `override` 而被指定, 如下例所示:

```
// Author A
namespace A
{
    public class Base                // version 2
    {
        public virtual void F() { // added in version 2
            System.Console.WriteLine("Base.F");
        }
    }
}

// Author B
namespace B
{
    class Derived: A.Base            // version 2b: override
    {
        public override void F() {
            base.F();
            System.Console.WriteLine("Derived.F");
        }
    }
}
```

```

    }
}
}

```

Derived 的作者还有另一个选择，那就是改变 F 的名字，这样就完全避免了名字冲突。尽管这个变化将打破 Derived 的原兼容性和二进制兼容性，这种相容的重要性随不同情况的变化而改变。如果 Derived 不被暴露给其他程序，那么，改变 F 的名字可能就是一个好方法。因为它将改善该程序的可读性—将不再会混淆 F 的含义。

1.14 属性 (Attributes)

C#是一种程序语言，但与其他程序语言相似，它也具有一些声明元素。例如，类中方法的可访问性通过其修饰语如 public, protected, internal, protected internal 或 private 而被指定。通过其对属性的支持，C#具有了这个能力，这使得程序员能够发明新的声明信息、为不同的程序实体指定这些声明信息，并在运行期恢复它们。程序通过定义和使用属性指定这些额外的声明信息。

例如，一个 framework 可以定义一个 Help Attribute 属性，这个属性可被放在程序元素如类和方法中，这使得开发者能够绘制一个从程序元素到文献的映射。下面的例子定义一个属性类 Help Attribute，简称 Help:

```

using System;

[AttributeUsage(AttributeTargets.All)]

public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Topic = null;
    private string url;

    public string Url {
        get { return url; }
    }
}

```

它具有一个位置参数 (string Url) 和一个已命名的自变量(string Topic)。位置参数由属性类的公用读写特性定义。下面的例子:

```

[Help("http://www.mycompany.com/.../Class1.htm")]

public class Class1
{
    [Help("http://www.mycompany.com/.../Class1.htm", Topic = "F")]
    public void F() {}
}

```

就是属性的几个应用。

通过使用反射支持，一给定程序元素的属性信息可在运行期被恢复。下面的例子：

```
using System;

class Test
{
    static void Main() {
        Type type = typeof(Class1);
        object[] arr = type.GetCustomAttributes(typeof(HelpAttribute));
        if (arr.Length == 0)
            Console.WriteLine("Class1 has no Help attribute.");
        else {
            HelpAttribute ha = (HelpAttribute) arr[0];
            Console.WriteLine("Url = {0}, Topic = {1}", ha.Url,
ha.Topic);
        }
    }
}
```

检查 Class1 是否具有一个属性 Help，如果这个属性存在，就编写出相关的 Topic 和 Url 值。

第2章 词法结构

2.1 翻译阶段 (Phases of Translation)

一个 C# 程序包括一个或多个源文档 (source files)。一个源文档就是一系列按顺序排列的单词字母。源文档与文件系统中的文件一一对应,但这种对应不是必需的。

从概念上来说,程序的编辑有两步:

- 词法分析,它把输入字母翻译成标识符。
- 句法分析,它把标识符翻译成可执行的代码。

2.2 语法符号 (Grammar Notation)

这部分对 C# 中的词法和句法的语法进行解释。词法语法说明字符是怎样形成标识符的;句法语法则说明标识符是怎样形成 C# 程序的。

语法的产生过程包括非终端符号和终端符号。在语法产生过程中, **non-terminal** 出现在斜体类型中, **terminal** 符号出现在宽度固定的活字中。每一个非终端符号都有一套结果产生过程定义。一套结果产生过程的第一条线就是非终端的名字,其后一个冒号。每一个成功的缩进线都包括一个结果产生过程的右半部分,非终端符号为其左半部分。下面的例子:

```
nonsense:
terminal1
terminal2
```

说明 **nonsense** 非终端具有两个结果产生过程,一个是其右半部分的 **terminal1**,一个是在其左半部分的 **terminal2**。

可选项一般被列在单独的线中,但有很多可选项时,短语“其中的一个”就会出现在这些列表之前。这只是在各个线上所列出的每一个可选项的缩写形式。下面的例子:

```
letter: one of
A B C a b c
```

就是下面这个式子的缩写形式:

```
letter: one of
A
B
C
a
b
c
```

下标“opt”如在 identifieropt 中所示，就是一个缩写形式，它表示一个可选择的符号。下面的例子：

```
whole:
    first-part  second-partopt  last-part
```

就是下例的一个缩写：

```
whole:
    first-part  last-part
    first-part  second-part  last-part
```

2.3 词法分析 (Lexical Analysis)

词法分析阶段的任务是把输入字符翻译成输入元素。如果不止一个输入元素能够与下一列字符相匹配，那么，可能性最大的输入元素就被使用，无论其后面的字符或符号正确与否。例如，“5++6”总是由三个输入元素组成，即“5”、“++”、“6”，虽然从句法的角度来看，把它分解成“5”、“+”、“+”和“6”可能更准确。

有时输入字符中的每一个字符都只是一个输入元素的一部分，一旦一系列字符形成一个输入元素，它就很难被其它标识符重新浏览。例如，在一直译字符串字母符号 (2.4.4.5 节) 中，字符并不与预处理符号相匹配。

一旦字符构成输入元素，空格和注释就被取消，且预处理指令也被处理掉，只留下一串符号。句法和语义处理也只出现在一串标识符上。

2.3.1 输入 (Input)

```
input:
    input-elementsopt
    input-elements:
        input-element
    input-elements  input-element
    input-element:
        comment
        white-space
        pp-directive
        token
```

2.3.2 字符输入 (Input Characters)

在 C# 中提供的输入字符可以为：
任何单码单符

2.3.3 线终端函数 (Line Terminators)

在 C# 中提供的新的行界限如下:

- 运载返回字符 (U+000D)。
- 线反馈字符 (U+000A)。
- 后跟一线反馈字符的运载返回字符。
- 线分离函数数字符 (U+2028)。
- 段分离函数数字符 (U+2029)。

2.3.4 注释 (Comments)

C# 支持两种注释: 规则注释和单行注释。

一个规则注释 (regular comment) 开始于字符 /*, 结束于字符 */。规则注释可占有一条线、一个线或多条线的一定比例。下面的例子:

```
/* Hello, world program
    This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        Console.WriteLine("hello, world");
    }
}
```

就含有一个规则注释。

一个单行注释 (one-line comment) 以 // 开头, 延伸至该线的结束。下面的例子就是几个单行注释:

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        Console.WriteLine("hello, world");
    }
}

comment:
regular-comment
one-line-comment
```

```

regular-comment:
/*  rest-of-regular-comment
rest-of-regular-comment:
*  rest-of-regular-comment-star
not-star  rest-of-regular-comment
rest-of-regular-comment-star:
/*  rest-of-regular-comment-star
not-star-or-slash  rest-of-regular-comment
not-star:
Any input-character except *
not-star-or-slash:
Any input-character except * and /
one-line-comment:
/  /  one-line-comment-text  new-line
one-line-comment-text:
input-character
one-line-comment-text  input-character

```

2.3.5 空格 (White Space)

在 C# 中的空格包括:

- 新行。
- Tab 字符 (U+0009)。
- 垂直 Tab 字符 (U+000B)。
- 形式反馈字符 (U+000C)。
- “控制-Z” 或 “替换” 字符 (U+001A)。
- 所有字符都是单码类 Zs。

2.4 标识符 (Tokens)

标识符有下列几种: 识别符号、关键字、字母、操作符和标点符号。空格和注释都不是标识符, 尽管它们可能具有标识符的分隔符作用。

```

token:

identifier

keyword

literal

operator-or-punctuator

```

2.4.1 单码字符转义序列 (Unicode Character Escape Sequences)

一个单码字符转义序列表示一个单码字符。单码字符转义序列在识别符号、非 verbatim 串字母以及字符字母中被执行。它不在其它任何位置（如操作符、标点符号或关键字的形成过程）中被执行。

unicode-character-escape-sequence:

`\u` hex-digit hex-digit hex-digit hex-digit

`\U` hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
hex-digit hex-digit

一个单码字符转义序列表示由字符“\u”或“\U”之后的 hexadecimal 数字形成的单个单码字符。由于 C#在字符或字符串中使用单码字符的 16 位代码，一个范围在 U+10000 和 U+10FFFF 的单码字符可用两个单码“surrogate”字符来表示。C#不支持 x10FFFF 之外的具有代码点的单码字符。

C#不执行多个翻译。例如，字符串字母“\U005Cu005C”就相当于“\U005C”而不是“\\”。（单码值\U005C 是字符“\”）。下面的例子就是\U0066 的几个应用，即字母“f”的字符转义序列：

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            Console.WriteLine(c.ToString());
    }
}
```

这个程序就相当于：

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            Console.WriteLine(c.ToString());
    }
}
```


2.4.2 标识符 (Identifiers)

这些标识符规则与被标准的 Unicode3.0、Technical Report15、Annex7 重新调用的一致。除下划线可作为初始化符号外（如在传统的 C 程序语言中），单码字符可出现在识别符中，并且，符号“@”就是关键字作为标识符的前缀。

```

identifier:
available-identifier
@identifier-or-keyword
available-identifier:
An identifier-or-keyword that is not a keyword
identifier-or-keyword:
identifier-start-character  identifier-part-charactersopt
identifier-start-character:
letter-character
_ (the underscore character)
identifier-part-characters:
identifier-part-character
identifier-part-characters  identifier-part-character
identifier-part-character:
letter-character
decimal-digit-character
connecting-character
combining-character
formatting-character
letter-character:
A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
A unicode-character-escape-sequence representing a character of classes
Lu, Ll, Lt, Lm, Lo, or Nl
combining-character:
A Unicode character of classes Mn or Mc
A unicode-character-escape-sequence representing a character of classes
Mn or Mc
decimal-digit-character:
A Unicode character of the class Nd
A unicode-character-escape-sequence representing a character of the class
Nd
connecting-character:
A Unicode character of the class Pc

```

A unicode-character-escape-sequence representing a character of the class Pc
 formatting-character:
 A Unicode character of the class Cf
 A unicode-character-escape-sequence representing a character of the class Cf

合法的标识符如“identifier1”、“-identifier2”以及“@if”。

作为标识符的关键字的前缀为“@”，这个前缀符号有利于与其它程序语言的联系。字符@实际上不是标识符的一部分，因此，在其它语言中经常作为一种没有前缀的标识符而出现。前缀@也可以用在不是关键字的标识符中，但这种风格很少用（不受欢迎）。下面的例子定义一个类“class”，这个类含有一个带有参数“bool”的静态方法“static”：

```
class @class
{
    static void @static(bool @bool) {
        if (@bool)
            Console.WriteLine("true");
        else
            Console.WriteLine("false");
    }
}

class Class1
{
    static void M {
        @class.@static(true);
    }
}
```

如果两个标识符按顺序进行下面的转化之后是一致的，那么，就认为这两个标识符相同：

- 除去已用过的前缀“@”。
- 把每一个 unicode-character-escape-sequence 都转化为相应的单码字符。

以两个连续的下划线字符开头的标识符被留作执行时用，且不被普通程序使用而被重新设定。例如，一个执行可能因以两个下划线开头的关键字的使用而被延伸。

2.4.3 关键字 (Keywords)

一个 keyword 就是一个备用的类似于标识符的字符序列，并且，如果没有前缀@，它不能作为标识符。C#中的关键字如表 2-1 所示。

表 2-1 关键字

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	lock	long	namespace	while
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void

在语法的某些位置，特定的标识符具有特定的意义，但它们不是关键字。例如，在一个属性声明过程中，标识符“get”和“set”具有特殊的含义。在这些位置，一个标识符绝对不是一个有效符号。因此，这与它们用作标识符并不矛盾。

2.4.4 字母 (Literals)

数值是一个值的源代码表示过程。C#中的数值类型包括下面几种：

- (布尔型数值) Boolean-Literal;
- (整型数值) Integer-Literal;
- (实数型数值) Real-Literal;
- (字符型数值) Character-Literal;
- (字符串) String-Literal;
- (null 数值) Null-Literal。

2.4.4.1 布尔型字母 (Boolean Literals)

布尔型字母值有两个：true 和 false。

```
boolean-literal:
true
false
```

boolean-literal 的类型是 bool。

2.4.4.2 整型字母 (Integer Literals)

整型字母有两种可能形式：十进制和十六进制。

```
integer-literal:
decimal-integer-literal
hexadecimal-integer-literal

decimal-integer-literal:
decimal-digits integer-type-suffixopt

decimal-digits:
decimal-digit
decimal-digits decimal-digit

decimal-digit: one of
0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of
U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal:
0x hex-digits integer-type-suffixopt
0X hex-digits integer-type-suffixopt

hex-digits:
hex-digit
hex-digits hex-digit

hex-digit: one of
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
```

整型字母类型的确定方式如下：

- 如果这个字母没有后缀,那么,它就具有这些类型中的第一种,其值可表示为:int、uint、long、ulong。
- 如果这个字母具有后缀 U 或 u,那么,它就具有下列类型中的第一种,其值可表示为:uint、ulong。
- 如果这个字母具有后缀 L 或 l,那么,它就具有下列类型中的第一种,其值可表示为:long、ulong。
- 如果这个字母具有后缀 UL,Ul,Ul,ul,LU,Lu,Lu 或 lu,那么,其类型就是 ulong。整型字母所表示的值不能在类型 ulong 的范围之外,否则,就是错误的。在表示类型 long 时,一般用“L”,而不用“l”,因为“l”易与“1”相混淆。为把 int 和 long 的最小值以十进制整数类型书写,有下列两条规则:

1) 当一个其值为 2^{31} ,且没有 integer-type-suffix 的 decimal-integer-literal 作为一个识别符号出现在一元减操作符(7.6.2 节)后面时,其结果就是一个具有值 -2^{31} 的常量。在所有其他情况下,这样的 decimal-integer-literal 的类型都是 uint。

2) 当一个具有值 $9223372036854775808 (2^{63})$ ，且没有 `integer-type-suffix` 或 `integer-type-suffixL` 或 `l` 的 `decimal-integer-literal` 作为一个识别符号出现在一元减操作符之后时，其结果就是一个具有值 $-9223372036854775808 (-2^{63})$ ，类型为 `long` 的常量。在所有其他情况下，这样的 `decimal-integer-literal` 的类型都是 `ulong`。

2.4.4.3 实数型字母 (Real Literals)

```
real-literal:
decimal-digits . decimal-digits exponent-partopt real-type-
suffixopt
. decimal-digits exponent-partopt real-type-suffixopt
decimal-digits exponent-part real-type-suffixopt
decimal-digits real-type-suffix
exponent-part:
e signopt decimal-digits
E signopt decimal-digits
sign: one of
+ -
real-type-suffix: one of
F f D d M m
```

如果没有实数型后缀，那么，实数型字母的类型就是 `double`。否则，实数型后缀决定实数型字母的类型，如下所示：

- 后缀为 `F` 或 `f` 的实数型字母的类型是 `float`。例如，字母 `1f`, `1.5f`, `1e10f` 以及 `123.456F` 的类型都是 `float`。
- 后缀为 `D` 或 `d` 的实数型字母的类型为 `double`。例如，字母 `1d`, `1.5d`, `1e10d` 以及 `123.456D` 的类型都是 `double`。
- 后缀为 `M` 或 `m` 的实数型字母的类型为 `decimal`。例如，字母 `1m`, `1.5m`, `1e10m` 以及 `123.456M` 的类型都是 `decimal`。

如果某一字母不能用所给类型表示，那么，就会出现编译错误。

2.4.4.4 字符型字母 (Character Literals)

一个字符型字母表示一个字符，且这个字符通常是带有引号的，如 `'a'`。

```
character-literal:
' character '
character:
single-character
simple-escape-sequence
hexadecimal-escape-sequence
unicode-character-escape-sequence
```

single-character:

Any character except ' (U+0027), \ (U+005C), and new-line

simple-escape-sequence: one of

' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:

\x hex-digit hex-digitopt hex-digitopt hex-digitopt

在一个 character 中，位于小括号 (\) 之后的字符必须是下列子分之一：'，"，\，\n，\r，\t，\f，\b，\a，\0，\x。否则，编译时就会出现错误。

一个十六进制转义序列表示一个单码字符，其由“\x”加上一个十六进制数字构成。

一个简单的转义序列表示一个单码字符代码，如下表 2-2 所示。

表 2-2 转义序列

转义序列	字符名字	Unicode 编码
'	Single quote	0x0027
"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

Character-literal 的类型是 char。

2.4.4.5 字符串字母 (String Literals)

C#支持两种字符串字母：规则字符串字母和直译字符串字母。规则字符串字母包括零或多个双引号内的字符，如“hello,world”，且它可能即含有简单转义序列（如列表字符 \t），又含有十六进制转义字符。

直译字符串字母以字符@开头，其后是一个双引号字符、零或多个字符以及双引号内的另一个字符。一个简单的例子就是@“hello,world”。在一个直译字符串字母内，两个分隔符之间的字符被逐字翻译，quote-escape-sequence 除外。尤其是简单的转义序列以及十六进制转义序列不在直译字符串字母内处理。一个直译字符串字母可能涉及多行。

string-literal:

regular-string-literal

verbatim-string-literal

regular-string-literal:

" regular-string-literal-charactersopt "

```

regular-string-literal-characters:
regular-string-literal-character
regular-string-literal-characters  regular-string-literal-character
regular-string-literal-character:
single-regular-string-literal-character
simple-escape-sequence
hexadecimal-escape-sequence
unicode-character-escape-sequence
single-regular-string-literal-character:
Any character except " (U+0022), \ (U+005C), and new-line
verbatim-string-literal:
@" verbatim -string-literal-charactersopt  "
verbatim-string-literal-characters:
verbatim-string-literal-character
verbatim-string-literal-characters  verbatim-string-literal-character
verbatim-string-literal-character:
single-verbatim-string-literal-character
quote-escape-sequence
single-verbatim-string-literal-character:
any character except "
quote-escape-sequence:
""

```

在 `regular-string-literal-character` 内小括号 () 之后的字符必须是下列字符之一: ‘, “, \, o, a, b, f, n, r, t, U, x, v。否则, 编译时就会出现错误。下面的例子:

```

string a = "hello, world";           // hello, world
string b = @"hello, world";          // hello, world

string c = "hello \t world";          // hello      world
string d = @"hello \t world";         // hello \t world

string e = "Joe said \"Hello\" to me"; // Joe said "Hello"
string f = @"Joe said ""Hello"" to me"; // Joe said "Hello"

string g = "\\server\\share\\file.txt"; // \\server\\share\\file.txt
string h = @"\\server\\share\\file.txt"; //
\\server\\share\\file.txt

string i = "one\ntwo\nthree";
string j = @"one
two
three";

```

就是一系列字符串字母。最后一个字符串字母 `j` 是一个直译字符串字母，它涉及多行。两个引号之间的字符，包括空格如新的一行，都仍然是直译的。

一个 `string-literal` 的类型是 `string`。

没必要每一个字符串字母都产生一个新的字符串实例。当两个或多个从字符串相等操作符（7.9.7 节）来看相等的字符串字母出现在同一个程序中时，这些字符串指的都是同一个字符串实例。例如，下面程序的输出结果是 `True`，因为这两个字母指的是同一个字符串实例。

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        Console.WriteLine(a == b);
    }
}
```

2.4.4.6 null(空)字母 (The Null Literal)

`null-literal`:
`null`
`null-literal` 的类型是 `null`。

2.4.5 操作符与标点符号 (Operators And Punctuators)

操作符和标点符号都有几种类型。操作符被用在表达式中以说明含有一个或多个操作数的操作。例如，表达式 `a+b` 用操作符 `+` 把两个操作数 `a` 和 `b` 相加。标点符号的作用是分组或分隔。例如，标点符号 `“:”` 的作用是分隔语句列表中的语句。

operator-or-punctuator: one of

{ }	[]	()	.	,	;	:	+	-	*
/	%	&		^	!	~	=	<	>
?	++	--	&&		<<	>>	==	!=	<=
>=	+=	-=	*=	/=	%=	&=	=	^=	<=
>>=	->								

2.5 预处理指令 (Pre-processing Directive)

C#中的术语“预处理指令”只是为了与 C 程序语言统一起来。在 C#中，没有单独的预处理步骤；预处理指令的操作是词法分析阶段的一部分。

预处理指令总是以字符‘#’开头，它必须在该行的开头，空格除外。空格可出现在‘#’和下列标识符之间。

```
pp-directive:
pp-declaration
pp-conditional-compilation
pp-line-number
pp-diagnostic-line
pp-region
```

对于指令 `pp-declaration`, `pp-conditional-compilation`, 以及 `pp-line-number` 来说, 根据一般的规则, 该行剩余部分被进行词法分析, 忽略注释和空格。一个输入元素 (如一个规则注释) 在该行的结束点未终止是合法的。

对于指令 `pp-diagnostic-line` 和 `pp-region` 来说, 该行剩余部分不进行词法分析。

2.5.1 预处理标识符 (Pre-processing Identifiers)

预处理标识符所用的语法与规则的 C# 标识法所用的语法相似:

```
pp-identifier:
```

An identifier-or-keyword that is not true or false

符号 `true` 和 `false` 是不合法的预处理标识符, 因此, 不能用 `#define` 定义或用 `#undef` 取消定义。

2.5.2 预处理表达式 (Preprocessing Expressions)

操作符 `!`, `==`, `!=`, `&&` 以及 `!!` 允许出现在预处理表达式中。预处理表达式中圆括号的作用是进行分组。预处理表达式的编译期赋值所用的规则与布尔表达式的运行期赋值所用的规则相同。已“定义”的标识符被赋值为 `true`, 否则为 `false`。

```
pp-expression:
pp-equality-expression
pp-primary-expression:
true
false
pp-identifier
(pp-expression)
pp-unary-expression:
pp-primary-expression
! pp-unary-expression
pp-equality-expression:
pp-equality-expression == pp-logical-and-expression
pp-equality-expression != pp-logical-and-expression
```

```
pp-logical-and-expression:
pp-unary-expression
pp-logical-and-expression && pp-unary-expression
pp-logical-or-expression:
pp-logical-and-expression
pp-logical-or-expression || pp-logical-and-expression
```

2.5.3 预处理声明 (Pre-processing Declarations)

在预处理中，名字可被定义或取消定义。一个 `#define` 定义在一个文档范围内的标识符。一个 `#undef` “取消定义” 一个文档范围内的标识符——如果这个标识符被定义得较早，那么，它就变成未被取消定义。如果一个标识符被定义，那么，从语义上来讲它就相当于 `true`；如果一个标识符被取消定义，那么，从语义上来讲它就相当于 `false`。

注意：用 `#define` 定义一个预处理标识符不影响该标识符在预处理指令之外的任何应用。

```
pp-declaration:
# define pp-identifier new-line
# undef pp-identifier new-line
```

下面的例子：

```
#define A
#undef B

class C
{

    #if A
        void F() {}
    #else
        void G() {}
    #endif

    #if B
        void H() {}
    #else
        void I() {}
    #endif
}
```

就成为：

```
class C
{
    void F() {}
    void I() {}
}
```

一个 **pp-declaration** 必须在输入文档的任何象征符号之前出现。也就是说，**#define** 和 **#undef** 必须出现在文档的任何“实数型代码”之前，否则，就会出现编译错误。因此，可以将 **#if** 和 **#define** 如下例分布：

```
#define A
#if A
#define B
#endif
namespace N
{
    #if B
    class Class1 {}
    #endif
}
```

下面的例子是非法的，因为 **#define** 在实数型代码之后：

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

#undef 可以“取消定义”一个未定义的名字。下面的例子定义了一个名字，然后对其进行两次取消定义；第二个 **#undef** 没有效果，但仍是合法的。

```
#define A
#undef A
#undef A
```

2.5.4 #if, #elif, #else, #endif

```
pp-conditional-compilation:
# if pp-expression new-line
# elif pp-expression new-line
```

```
# else    new-line    groupopt
# endif   new-line
```

一系列 `pp-conditional-compilation` 指令的作用是有条件地包括或排除程序文的某些部分。指令 `pp-conditional-compilation` 必须按顺序出现，如下所示：一个 `#if` 指令，零或多个 `#elif` 指令，零或多个 `#else` 指令，一个 `#endif` 指令。指令 `pp-conditional-compilation` 的集合可以嵌套，只要在其上属集合的两个指令之间有一个完整的集合。

当对一批 `pp-conditional-compilation` 指令被操作时，两个指令之间的某些上下文就被包括，或者说在进行词法分析时，有些则被排除在外，也就是说不再对其进行进一步处理。被包括或排除的上下文严格地位于两个 `pp-conditional-compilation` 指令之间。受影响的上下文在对其进行词法分析之前被包括或排除；如果被包括，那么，它就不能被输入元素，如注释、字母或其它象征符号浏览。

包括还是排除上下文根据对指令中表达式的赋值结果而定，如下所示：

- 如果 `#if` 指令上的 `pp-expression` 赋值为 `true`，那么，这个集合中 `#if` 同下一个 `pp-conditional-compilation` 指令之间的上下文就被包括，在指令 `#if` 和 `#endif` 之间的所有其它上下文就被排除。

- 否则，如果所有的 `#elif` 指令都存在，那么，与它们有关的 `pp-expression` 就被按顺序求值。如果每一个求值结果都是 `true`，那么，在第一个求值结果为真的 `#elif` 指令与下一个 `pp-conditional-compilation` 指令之间的上下文就被包括，在指令 `#if` 和 `#endif` 之间的所有其它上下文都被排除。

- 否则，如果存在一个指令 `#else` 那么，在指令 `#else` 和 `#endif` 之间的上下文就被包括，在指令 `#if` 和 `#endif` 之间的所有其它上下文都被排除。

- 否则，在指令 `#if` 和 `#endif` 之间的所有上下文都被排除。

下面的例子：

```
#define Debug

class Class1
{
    #if Debug
        void Trace(string s) {}
    #endif
}
```

变成：

```
class Class1
{
    void Trace(string s) {}
}
```

如果组成部分可被嵌套。例如：

```

#define Debug          // Debugging on
#undef Trace           // Tracing off

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}

```

那么，不被排除的上下文就不进行词法分析。例如，下面的例子就是合法的，尽管在“#else”那部分存在无终止的注释：

```

#define Debug          // Debugging on

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}

```

预处理指令如果出现在输入元素内，那么，它们就不能被处理。例如，下面的程序：

```

class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
world
#else
Nebraska
#endif

```

```
        ");  
    }  
}
```

输出:

```
hello,  
#if Debug  
world  
#else  
Nebraska  
#endif
```

2.5.5 #error 和 #warning

指令 **#error** 和 **#warning** 用代码向编译器报告错误和警告的有关情况，以便于标准的编译错误和警告统一起来。

```
pp-diagnostic-line:  
# error pp-message  
# warning pp-message  
pp-message:  
pp-message-characters new-line  
pp-message-characters:  
pp-message-character  
pp-message-characters pp-message-character  
pp-message-character:  
Any character except new-line
```

下面的例子:

```
#warning Code review needed before check-in  
  
#define DEBUG  
  
#if DEBUG && RETAIL  
#error A build can't be both debug and retail!  
#endif  
  
class Class1  
{...}
```

总是产生一个警告“Code review needed before check-in”，且如果预处理标识符 **debug** 和 **retail** 都被定义，就产生一个错误。

2.5.6 #region 和 #endregion

```
pp-region:
# region    pp-message
# endregion pp-message
```

指令 `#region` 和 `#endregion` 指定了代码的“范围”（区域）。区域在语义上没有意义，程序员或自动工具用它们标记一个代码。与 `#region` 和 `#endregion` 有关的信息也没有语义意义，它只是用于标识一个区域。每一个源文档都必须具有数目相等的 `#region` 和 `#endregion` 指令，且在该文档中每一个 `#region` 之后都必须具有一个相匹配的 `#endregion`。

`#region` 和 `#endregion` 都必须根据条件编译指令正确地嵌套。确切地说，一个 `pp-conditional-compilation` 指令出现在 `#region` 及其匹配指令 `#endregion` 之间是错误的，除非这个集合中所有的 `pp-conditional-compilation` 指令都出现在 `#region` 和 `#endregion` 之间。

2.5.7 #line

`#line` 使得开发者能够改变在输出警告和错误时编译器所用的行数和源文档名。如果没有行指令，那么，行数和文档名就由编译器自动确定。指令 `#line` 一般被用于从某些其它程序输入中产生 C# 源代码的编程之后的工具中。在 `#line` 指令之后，编译器就认为在此指令之后的行都具有给定的行数（及文档名，如果被指定）。

```
pp-line-number:
# line integer-literal
# line integer-literal file-name

file-name:
" file-name-characters "

file-name-characters:
file-name-character
file-name-characters file-name-character

file-name-character:
Any character except " (U+0022), and new-line
```

注意：file-name 不同于普通字符串字母之处就在于转义字符不被处理；字符 “\” 只指定 file-name 内的一个普通的后置字符。

第3章 基本概念

3.1 程序开始 (Program Startup)

程序开始指的是执行环境调用一指定方法，即所谓的程序的 entry point。这个入口方法的名称为 Main，其用法有以下几种：

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

由此可见，入口可能返回一个 int 值。这个返回值表示程序结束 (3.2 节)。

入口可带有一形参，这个形参的名字可任意选取。但如果要声明这样一个参数，必须遵循下列规则：

(1) 这个参数的值不能是 null。

(2) 假设这个参数的名字是 args，如果由 args 指定的数组长度大于零，则其中 args[args.length-1] 的数组项 args[o] 必须指的是字符串，即 program parameters。这些参数的值在程序开始之前已由系统定义。其目的是为了给程序提供来自于系统其它部分、在程序开始之前已确定的信息。如果系统不能提供含字母大小写的字符串，应确保执行过程中小写字符串能够被识别。

(3) 由于 C# 支持方法重载，因此，在某一程序或结构中可对某些方法进行多次定义，但每次都有不同的签名。然而在一个程序中，程序入口的 Main 方法只有一个。如果程序中含有的参数多于一个或它们唯一的参数类型不是 string[]，那么，只有 Main 方法允许重载。

(4) 一个程序可以有多个类型或结构，两个或两个以上的这些类型和结构共用一个 Main 方法，这个 Main 方法用在程序入口。在任何情况下，必须有一个 Main 方法用在程序入口，以确保程序开始。至于如何选择入口，这里不再赘述。

(5) 在 C# 中，每一个方法都有多个类型或结构。通常情况下，一个方法的访问能力 (参见 3.5.1 节) 由其声明过程中所指定的访问修改函数 (参见 10.2.3 节) 来决定。同样，一个类型的访问能力也由其声明过程中所指定的访问修改函数来决定。要使某一给定类型的指定方法能够被调用，其类型或成员必须是可访问的。然而，程序入口异常，执行环境可直接访问程序入口，而不必考虑其访问能力及其类型声明过程的访问能力。

(6) 在其它方面，入口方法的使用同非入口方法一样。

3.2 程序结束 (Program Termination)

程序结束将控制返回到执行环境。

如果程序入口方法的返回类型是 `int`, 则这个返回值就表示程序的终止状态代码。这个代码的作用是把执行情况返回到执行环境。

如果入口方法的返回类型是 `void`, 则当执行过程中遇到一终止那个命令的右弧 “}” 或处于没有表达语句的 `return` 状态时, 结果为终止状态代码 0。

在一个程序结束之前, 除非其子程序的结束过程已被忽略, 否则, 它们的结束函数将首先被调用。至于结束语句可被忽略的情况, 这里不再赘述。

3.3 声明 (Declarations)

C#程序中的声明指的是对程序中的数组或元素进行定义。C#程序是由名字空间组成的, 名字空间包括类型声明和嵌套名字的空间声明。类型声明的作用是定义类、结构、接口、枚举和委托。类型声明的成员是由其形式决定的。例如, 类声明包括构造函数、析构函数、静态构造函数、常量、域、方法、属性、事件、索引、操作符和嵌套类型。

声明就是在其所属的 `declaration space` 内定义一个名字。除重载的构造函数、方法、索引以及操作符名称外, 在一个声明空间内, 用同一个名字声明两个或两个以上的成员是错误的。在一个声明空间内含有用同一个名字命名的不同种类的成员也是不可能的。例如, 在一个声明空间内, 绝对不能含有用同一个名字命名的域和方法。

以下是声明空间的几个不同的类型:

在一个程序的所有源文档中, `namespace-declarations` 内的 `namespace-member-declarations` 不可能是一个联合声明空间 `global declaration space` 的成员。

在一个程序的所有文档中, 具有相同全权名字空间的 `namespace-declaration` 中的 `namespace-member-declaration` 是一个联合空间的成员。

每一个类、结构或接口的声明都会产生一个新的声明空间。这类声明空间中所引用的名字分别来自于 `class-member-declarations`, `struct-member-declarations` 或 `interface-member-declarations`。除重载的构造函数声明和静态构造函数声明外, 类或结构的成员声明不能引入一个同这个类或结构的名字相同的成员。类、结构或接口允许重载的构造函数或操作符的声明。例如, 在一个类、结构或接口中可以含有多个用同一名字命名的方法声明, 只要这些方法声明的签名不同 (参见 3.6 节)。

注意: 其类型不产生类的声明空间, 基本接口也不产生接口的声明空间。因此, 派生的类或接口就可以用与继承成员相同的名字进行声明。那么, 就说这样的成员隐含继承成员。

- 每一个枚举声明产生一个新的声明空间。这类声明空间的名字来源于 `enum-member-declarations`。
- 每一个 `block` 或 `switch-block` 产生局部变量的独立的声明空间。这类声明空间所引用的名字来自于 `local-variable-declaration`。如果一个块是构造函数或方法声明的主体, 那么, 在 `formal-parameter-list` 中声明的参数就是这个块的 `local variable declaration space` 成员。一个块的局部变量的声明空间包括任何一个嵌套块。这样在一个嵌套块内, 声明一个同包含它的块的局部变量名字相同的局部变量是不可能的。
- 每一个 `block` 或 `switch-block` 产生一个独立的声明空间。这类声明空间中所引用的名字来自于 `labeled-statements`, 且这些名字是通过 `goto-statements` 而被引用的。一个块的

label declaration space 包括任何一个嵌套块。因此，在一个嵌套块内，声明一个同包含它的块中的符号具有相同名字的局部变量是不可能的。

名字声明的原文顺序通常并不重要，尤其是对于名字空间、类型、常量、方法、属性、事件、索引、操作符、构造函数、析构函数以及静态构造函数的声明及应用来说。但在下列情况下，声明顺序还是很重要的：

- 域及局部变量的声明顺序决定它们初始化（如果有）的执行顺序。
- 局部变量必须在使用之前被定义（参见 3.7 节）。
- 如果 constant-expression 的值可以忽略，则枚举成员的声明（0）顺序还是很重要的。

名字空间的声明空间“结果是通用的”，两个具有相同全权名字的名字空间声明产生相同的声明空间。例如：

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

上面的两个名字空间声明产生相同的声明空间，在这种情况下，分别用全权名字 `Megacorp.Data.Customer` 和 `Megacorp.Data.Order` 声明这两个类。由于这两个声明产生相同的声明空间，其类的声明名字就不能相同，否则，就是错误的。

一个块的声明空间包括任何嵌套块。因此，在下例中，方法 F 和 G 是错误的。因为在块外声明的名字 i 不能在块内重新被声明。而方法 H 和 I 是有效的，因为其中的两个 i 是分别在不同的嵌套块内被声明的。

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }
}
```

```
void G() {
    if (true) {
        int i = 0;
    }
    int i = 1;
}

void H() {
    if (true) {
        int i = 0;
    }
    if (true) {
        int i = 1;
    }
}

void I() {
    for (int i = 0; i < 10; i++)
        H();
    for (int i = 0; i < 10; i++)
        H();
}
```

3.4 元素 (Members)

名字空间和类型都有元素 (members)。只要使用合适的名字，实体的元素都是有效的。这种名字以实体开头，后面是符号 “.”，再后面便是元素的名字。

类型元素有的是在这个类型中声明的，有的是从这一类型的基类 inherited 而来的。当某一类型是从基类继承而来时，基类的所有元素（构造函数和析构造函数除外）都是所派生类型的元素。基类成员声明的可访问性并不能决定这个成员能否被继承——继承涉及除构造函数和析构造函数之外的任何元素。然而，在派生类型中，继承元素可能不能被访问。这可能是由其声明的可访问性（参见 3.5.1 节）造成的，也可能是因其被自身类型的一个声明所隐藏的缘故。

3.4.1 名字空间元素 (Namespace Members)

名字空间包含的名字空间和类型是 global namespace 的元素。这同总声明空间内声明的名字是一致的。

在一个名字空间内声明的名字空间和类型是那个名字空间的元素。这同在那个名字空间

的声明空间内所声明的名字是一致的。

名字空间没有访问限制。要声明局部的、被保护的或者内部的名字空间是不可能的。名字空间的名字总是可公开访问的。

3.4.2 结构成员 (Struct Members)

一个结构的成员就是指在这个结构中声明的成员以及从基本 object 中继承的成员。简单类型的成员同别名与其相同的结构类型的成员是一致的。

- sbyte 的成员就是结构 System.SByte 的成员。
- byte 的成员就是结构 System.Byte 的成员。
- short 的成员就是结构 System.Int16 的成员。
- ushort 的成员就是结构 System.UInt16 的成员。
- int 的成员就是结构 System.Int32 的成员。
- uint 的成员就是结构 System.UInt32 的成员。
- long 的成员就是结构 System.Int64 的成员。
- ulong 的成员就是结构 System.UInt64 的成员。
- char 的成员就是结构 System.Char 的成员。
- float 的成员就是结构 System.Single 的成员。
- double 的成员就是结构 System.Double 的成员。
- decimal 的成员就是结构 System.Decimal 的成员。
- bool 的成员就是结构 System.Boolean 的成员。

3.4.3 枚举成员 (Enumeration Members)

一个枚举的成员指的是在这个枚举中声明的常量以及从类 object 中继承的成员。

3.4.4 类项 (Class Member)

一个类的项包括在这个类中声明的项以及从基类（除类 object 外，因为它不含基类）中继承的项。从基类中继承的项有：常量、域、方法、属性、事件、索引、操作符及基类的类型，但不包括基类的构造函数、析构函数和静态构造函数。基类的项可被继承而不必考虑其可访问性。

一个类的声明可能包括常量、域、方法、属性、事件、索引、操作符、构造函数、析构函数、静态构造函数以及类型的声明。

object 和 string 的项同别名与其相同的类 type 的项一致。

- object 的项就是类 System.Object 的项。
- string 的项就是类 System.String 的项。

3.4.5 接口成员 (Interface Members)

一个接口的成员包括在这个接口或其所有基本接口中声明的成员以及从类 `object` 中继承的成员。

3.4.6 数组项 (Array Members)

一个数组的项指的是从类 `System.Array` 中继承的项。

3.4.7 委托成员 (Delegate Members)

委托成员指的是从类 `System.Delegate` 中继承的成员。

3.5 成员访问 (Member Access)

成员声明控制着成员访问。成员的可访问性是由这个成员声明的可访问性 (参见 3.5.1 节) 及其直属类型 (如果有) 的可访问性共同决定的。

当某一成员允许被访问时, 就说这个成员是 `accessible`。当某一成员不允许被访问时, 就说这个成员是 `inaccessible`。只有当访问的原文地址包含在这个成员的可访问域 (参见 3.5.2 节) 内时, 这个成员才可被访问。

3.5.1 声明的可访问性 (Declared Accessibility)

一个成员的 **declared accessibility** 有以下几种情况:

- 开放型, 它的选择包括成员声明中的一个 `public` 修改函数, `public` 的字面意思就是“访问没有限制”。
- 内部保护型 (即被保护的或内部的), 它的选择包括成员声明中的一个 `protected` 修改函数和一个 `internal` 修改函数。 `protected internal` 的字面意思就是“访问只限制在从其所包含的类中派生的程序或类型”。
- 保护型, 它的选择包括成员声明中的一个 `protected` 修改函数。 `protected` 的字面意思就是“访问只限制在从所包含的类中派生的类或类型”。
- 内部型, 它的选择包括成员声明中的一个 `internal` 修改函数。 `internal` 的字面意思就是“访问只限制在这个程序内”。
- 局部型, 它的选择包括成员声明中的一个 `private` 修改函数。 `private` 的字面意思就是“访问只限制在包含它的类型内”。

从有关成员声明的上下文可以看出, 只有声明为可访问的某些类型允许成员声明。而且, 当一个成员声明不含有任何访问修改函数时, 声明过程就会产生默认的声明的可访问性。

- 名字空间很明显具有 `public` 声明的可访问性。在名字空间声明内, 不允许有访问修改函数。

- 在编辑类中声明的类型或者名字空间可能具有 `public internal` 声明的可访问性，且默认为 `internal` 声明的可访问性。
- 类项可能具有声明可访问性的五种类型的任何一种，且默认为 `private` 声明的可访问性。（注：被声明为类项的类型可具有声明的可访问性五种类型的任何一种，而声明为名字空间成员的类型只能具有 `public` 或 `internal` 声明的可访问性。）
- 结构成员可具有 `public`、`internal` 或 `private` 声明的可访问性，默认为 `private` 声明的可访问性。结果成员不具有 `protected` 或 `protected internal` 声明的可访问性。
- 接口成员很明显具有 `public` 声明的可访问性。在接口成员声明中，不允许有访问修改函数。
- 枚举成员也具有 `public` 声明的可访问性。在枚举成员声明中，也不允许有访问修改函数。

3.5.2 可访问域 (Accessibility Domains)

成员的访问域 (accessibility domain) 是程序文的片段 (可能不连贯)，在程序文中，程序可被访问。为了定义成员的可访问域，如果它未在某一类型中被声明，则将被放在程序开始，如果它在另一类型中被声明，则将被嵌套。程序的程序文指的是这个程序的所有源文档包含的程序文，类型的程序文指的是在这个类型的 `class-body`、`struct-body`、`interface-body` 或 `enum-body` 中开关符号 “{” 和 “}” 之间的所有程序文 (可能包括这个类型内的嵌套类型)。

预先定义的类型 (如 `object`、`int` 或 `double`) 的可访问域是无限的。

在程序 P 中声明的起始类型 T 的可访问域的情况如下：

- 如果 T 声明的可访问性是 `public`，则 T 的可访问域包括 P 的程序文及与 P 有关的任何程序。

- 如果 T 声明的可访问性是 `internal`，则 T 的可访问域就是 P 的程序文。

由此可知，起始类型的可访问域至少应是声明这个类型的程序的程序文。

在程序 P 的类型 T 中声明的嵌套成员 M 的可访问域的确定方法如下 (注意：M 本身就可能是一种类型)：

- 如果 M 声明的可访问性是 `public`，则其可访问域就是 T 的可访问域。
- 如果 M 声明的可访问性是 `protected internal`，则其可访问域就是 T 的可访问域同 P 的程序文以及在 P 外声明的 T 所派生的任何类型的程序文的交叉部分。
- 如果 M 声明的可访问性是 `protected`，则其可访问域就是 T 的可访问域同 T 的程序文以及 T 所派生的任何类型的交叉部分。
- 如果 M 声明的可访问性是 `internal`，则其可访问域就是 T 的可访问域同 P 的程序文的交叉部分。
- 如果 M 声明的可访问性是 `private`，则其可访问域就是 T 的程序文。

由此可知，嵌套成员的可访问域至少应是声明这个成员的类型程序文。而且，一个成员的可访问域总是小于声明这个成员的类型可访问域。

从理论上讲，当一类型或成员 M 被访问时，下面的步骤估计能确保访问被允许。

- 首先，如果 M 在一类型 (与编辑函数或名字空间对立) 内被声明，而那个类型又不可访问，则错误出现。

- 其次, 如果 M 是 **public**, 则访问被允许。
- 否则, 如果 M 是 **protected internal**, 而访问是在声明 M 的程序内发生的或者是在声明 M 的类所派生的一个类内通过派生的类型 (参见 3.5.3 节) 而发生的, 则访问被允许。
- 否则, 如果 M 是 **internal**, 而访问是在声明 M 的程序内发生的, 则访问被允许。
- 否则, 如果 M 是 **private**, 而访问是在声明 M 的程序内发生的, 则访问被允许。
- 否则, 类型或成员便是不可访问的, 错误出现。

在下面的例子中:

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

类和成员具有下列可访问域:

- A 和 A.X 的可访问域是无限的。
- Y,B.X,B.Y,B.C.X 和 B.C.Y 的可访问域是包含它的程序的程序文。
- Z 的可访问域就是 A 的程序文。
- B.Z 和 B.D 的可访问域就是 B 的程序文, 它包括 B.C 和 B.D 的程序文。
- B.C.Z 的可访问域就是 B.C 的程序文。
- B.D.X,B.D.Y 和 B.D.Z 的可访问域就是 B.D 的程序文。

这个例子说明, 成员的可访问域绝不比类似类型的可访问域大。例如, 即使 X 的所有成员都具有开放的声明的可访问性, 除 A.X 外, 其它成员的可访问域都要受到一类似类型的限制。

如 3.4 节所述, 除构造函数和析构函数外, 其它类的所有成员都被派生类型继承。这甚至包括一个基类的局部成员。然而, 一个局部成员的可访问域只包括声明该成员的类型程序文。在下例中类 B 从类 A 继承局部成员 x:

```
class A
{
    int x;

    static void F(B b) {
        b.x = 1;        // Ok
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;        // Error, x not accessible
    }
}
```

由于这个成员是局部的, 它只在 A 的 class-body 中具有可访问性。因此, 对 b.x 的访问在 A.F 方法中是可行的, 而在 B.F 方法中则是不可行的。

3.5.3 访问保护 (Protected Access)

当 `protected` 或 `protected internal` 的成员在声明它们的程序的程序文外被访问时, 访问的发生需要通过能使其发生的派生的类类型。设 B 为声明受保护成员 M 的基类, D 为 B 派生的一个类。则在 D 的 class-Body 内, 对 M 的访问有以下几种形式:

- 式子 M 的无权 `type-name` 或 `primary-expression`。
- 式子 T.M 的 `type-name`, 假设 T 是 D 或 D 派生的一个类。
- 式子 E.M 的 `primary-expression`, 假设 E 的类型是 D 或 D 派生的一个类。
- 式子 Base.M 的 `primary-expression`。

除访问这些式子外, 派生类还可以访问 `constructor-initializer` (参见 10.10.1 节) 的基类中受保护的构造函数。

在下例中:

```
public class A
{
    protected int x;
```



```

static void F(A a, B b) {
    a.x = 1;          // Ok
    b.x = 1;          // Ok
}
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;      // Error, must access through instance of B
        b.x = 1;      // Ok
    }
}

```

在 A 内通过 A 和 B 都可以访问 x，因为在这两种情况下，访问都是通过 A 或 A 派生的类而发生的。然而，在 B 内不可能通过 A 访问 B，因为 A 不是由 B 派生的。

3.5.4 访问约束 (Accessibility Constraints)

C#语言中有些构造函数需要这样一个类型，即其可访问性至少同成员或其它类型一致。如果类型 T 的可访问域比 M 的大，则类型 T 的可访问性至少同某一个成员类型 M 的可访问性相同。也就是说，如果 T 在 M 可访问的所有上下文中都可访问，则 T 的可访问性至少与 M 相同。

访问约束有以下几种情况：

- 直属于一类类型的基类至少具有该类类型本身的可访问性。
- 接口类型的基本接口至少具有接口类型本身的可访问性。
- 委托类型的返回类型和参数类型至少具有该委托类型本身的可访问性。
- 常量类型的可访问性至少应与常量本身的可访问性相同。
- 域类型的可访问性至少应与域本身的可访问性相同。
- 方法的返回类型和参数类型至少具有方法本身的可访问性。
- 属性类型的可访问性至少与属性本身的可访问性相同。
- 事件类型的可访问性至少与事件本身的可访问性相同。
- 索引类型及其参数类型的可访问性至少与索引本身相同。
- 操作符的返回类型和参数类型的可访问性至少与操作符本身相同。
- 构造函数的参数类型的可访问性至少与构造函数本身的可访问性相同。

在下例中：

```

class A {...}

public class B: A {...}

```

类 B 是错误的，因为 A 的可访性并非至少与 B 相同。

同样，在下例中：

```
class A {...}

public class B
{
    A F() {...}

    internal A G() {...}

    public A H() {...}
}
```

B 中的 H 方法也是错误的，因为 A 的返回类型并非至少与这个方法的可访问性相同。

3.6 签名和重载 (Signatures And Overloading)

方法构造函数索引和操作符因其签名的不同而不同：

- 方法的签名包括方法名及每个形参的类型和种类（值、引用或输出）。方法的签名特殊之处就在于它既不包括返回类型也不包括可能被上个参数所特化的 `params` 修改函数。
- 构造函数的签名包括每个形参的类型和种类（值、引用或输出）。构造函数的签名也不包括可能被上个参数所特化的 `params` 修改函数。
- 索引的签名包括每个形参的类型，但不包括成员类型。
- 操作符的签名包括操作符的名字以及每个形参的类型，但不包括结果类型。

签名使得类、结构以及接口成员的重载成为可能：

- 假设方法的签名都是独特的，则方法重载允许一个类、结构或接口用同一个名字声明多个方法。
- 构造函数的重载允许一个类或结构声明多个构造函数，假设构造函数的签名都是唯一的。
- 索引重载允许一个类或结构声明多个索引，假设索引的签名都是唯一的。
- 操作符允许一个类或结构用同一个名字声明多个操作符，假设操作符的签名都是唯一的。

下面的例子是一带有签名的重载方法声明过程：

```
interface ITest
{
    void F(); // F()
    void F(int x); // F(int)
    void F(ref int x); // F(ref int)
    void F(out int x); // F(out int)
    void F(int x, int y); // F(int, int)
    int F(string s); // F(string)
```

```

int F(int x);           // F(int)

void F(string[] a);     // F(string[])

void F(params string[] a); // F(string[])
}

```

注意: ref 和 out 参数修改函数是(参见 10.5.1 节)签名的一部分。因此, F(int)、F(ref int) 和 F(out int)都是唯一的签名。还要注意, 返回类型和 params 修改函数不是签名的一部分, 且只基于返回类型或修改函数之内或之外的重载是不可能的。由于这些限制, 在编辑上面的程序时, 可能会因方法带有重复签名 F(int)和 F(string[])而出现错误。

3.7 范围 (Scopes)

3.7.1 名字的范围 (Name Scopes)

名字的范围指的是程序文的区域, 在这个区域中可以查找任何名字声明的实体。范围可被嵌套, 且内部范围可以从外部范围重新声明一个名字的意思。外部范围的名字则被隐藏在被内部范围所覆盖的程序文的区域下面。对外部名字来说, 只有其被授权后才有可能访问。

- 无 namespace-declaration 包含的 namespace-member-declaration 所声明的名字空间成员的范围是每个编辑函数的整个程序文。
- 被全权名字为 N 的 namespace-declaration 内的 namespace-member-declaration 所声明的名字空间成员的范围是每一个全权名字为 n 或以与 n 相同的标识序列开头的 namespace-declaration 的 namespace-body。
- 由 using-directive 定义或引入的名字的范围并不只包括发生 using-directive 的 compilation-unit 或 namespace-body 的 namespace-member-declarations。一个 using-directive 可能会产生零或多个名字空间或在某些 compilation-unit 或 namespace-body 内已存在的类型名, 但它并不给潜在的声明空间提供任何新的成员。换句话说, using-directive 不能转移, 而只能影响发生它的 compilation-unit 或 namespace-body。
- 由 class-member-declaration 声明的成员的范围内是发生声明的 class-body。另外一个类成员的范围也包括在这个成员的可访问域(参见 3.5.2 节)内的派生的类的 class-body。
- 由 struct-member-declaration 声明的成员的范围内是发生这个声明的 struct-body。
- 由 enum-member-declaration 声明的成员的范围内是发生这个声明的 enum-body。
- 在 constructor-declaration 内声明的参数的范围是那个 constructor-declaration 的 constructor-initializer 和 block。
- 在 method-declaration 内声明的参数的范围是那个 method-declaration 的 method-body。
- 在 indexer-declaration 内声明的参数的范围是那个 indexer-declaration 的 accessor-declaration。
- 在 operator-declaration 内声明的参数的范围是那个 operator-declaration 的 block。

- 在 `local-variable-declaration` 内声明的局部变量的范围是这个声明发生的区域。在位于该局部变量的 `variable-declaration` 之前的文本位置查找局部变量是不正确的。
- 在一个 `for` 状态的 `for-initializer` 内声明的局部变量的范围是这个 `for` 状态的 `for-initializer`, `for-condition`, `for-iterater` 以及其所有的状态。
- 在 `labeled-statement` 内所声明的符号的范围是发生这个声明的 `block`。
- 在一个名字空间、类、结构或枚举成员的范围，不可能在文本位置查找到这个成员，因为文本位置位于这个成员声明之前。例如：

```
class A
{
    void F() {
        i = 1;
    }

    int i = 0;
}
```

这里，F 方法在 i 声明之前查找是有效的。

在局部变量的范围内，要在文本位置查找这个局部变量是错误的，因为文本位置在局部变量的 `variable-declarator` 之前。例如：

```
class A
{
    int i = 0;

    void F() {
        i = 1; // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = {j = 1}; // Legal
    }

    void H() {
        int a = 1, b = ++a; // Legal
    }
}
```

在上面的方法 F 中，对 i 的第一次分配不涉及在外部范围声明的域。它涉及到局部变量，但在这里是错误的，因为在上下文中它位于变量声明之前。在方法 G 中，声明 j 的初始化中 j 的运用是合法的，因为它的运用不在 `variable-declaration` 之前。在方法 F 中后来的 `variable-declarator` 涉及到在同样的 `local-variable-declaration` 内的较靠前的 `variable-declaration` 内声明的局部变量。

局部变量的范围划分规则确保在一个区内用于表达式上下文的某一个名字的意思是相同的。如果一个局部变量的范围只是从它的声明到这个区的结束,则上面的例子中第一次赋值指的是实例变量,第二次赋值指的是局部变量,如果区的状态后来被重新改变,则可能会导致错误的出现。

一个区内某一个名字的意思根据上下文的需要可以有所不同。在下面的例子中:

```
class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                                // expression context

        Type t = typeof(A);                          // type context

        Console.WriteLine(s);                        // writes "hello, world"
        Console.WriteLine(t.ToString());             // writes "Type: A"
    }
}
```

名字 *a* 在表达式上下文中指的是局部变量 *a*,而在类型上下文中指的是类 *a*。

3.7.2 名字隐藏 (Name Hiding)

实体的范围一般比其声明的空间所包含的程序文要多。特殊情况下,一个实体的范围可能会包含这样的声明,这些声明引入新的含有同样名字实体声明空间。这样的声明导致最初的实体被隐藏。如果实体不被隐藏就说它是可见的。

当范围与嵌套或继承重叠时就会出现名字隐藏。隐藏的两种类型的特征将在下面几部分进行说明。

3.7.2.1 通过嵌套隐藏 (Hiding Through Nesting)

通过嵌套的名字隐藏可能有以下几种操作结果,即嵌套名字空间或名字空间内的类型、嵌套类或结构内的类型以及参数和局部变量声明。范围通过嵌套的名字隐藏不易发觉,也就是说,当外部名字被内部名字隐藏时,没有错误或警告显示。

例如:

```
class A
{
    int i = 0;

    void F() {
        int i = 1;
    }
}
```

```
void G() {  
    i = 1;  
}  
}
```

方法 F 中，实例变量 i 被局部变量 i 隐藏，但在方法 G 中，i 仍然指的是实例变量。

当内部范围的一个名字隐藏外部范围的一个名字时，它将隐藏那个名字的重叠部分。例如：

```
class Outer  
{  
    static void F(int i) {}  
  
    static void F(string s) {}  
  
    class Inner  
    {  
        void G() {  
            F(1);                // Invokes Outer.Inner.F  
            F("Hello");          // Error  
        }  
  
        static void F(long l) {}  
    }  
}
```

对 F(1) 的调用产生在 Inner 内声明的 F，因为 F 的所有外部事件都被内部声明隐藏了。同样原因，对 (“hello”) 的调用是错误的。

3.7.2.2 通过继承隐藏 (Hiding Through Inheritance)

当类或结构重新声明从基类继承的名字时，就会出现通过继承的名字隐藏。这种类型的名字隐藏有以下几种形式：

- 类或结构中所引用的常量、域、属性、事件或类型隐藏具有相同名字的所有基类成员。
- 类或结构中所引用的方法隐藏所有具有相同名字的方法、基类项以及所有具有相同签名（方法名及参数数量、修改函数和类型）的基类方法。
- 类或结构中所引入的索引隐藏所有具有相同签名（参数数量和类型）的基类索引。

操作符的声明规则（参见 10.9 节）使得一个派生的类不可能声明一个具有相同签名的操作符来作为某基类的一个操作符。因此，操作符从不彼此隐藏。

与隐藏一个来自外部范围的名字相反，隐藏一个来自于被继承范围的可访问名字时有警告显示。例如：

```

class Base
{
    public void F() {}
}

class Derived: Base
{
    public void F() {}    // Warning, hiding an inherited name
}

```

Derived 中 F 的声明有警告显示。然而，隐藏继承名字并没有错，因为它将会防止基类的独立形成。例如，由于 Base 后来的版本引入了一个在这个类的早期版本中并不存在的方法 F，上面的情况就可能发生。如果上面的情况是错误的，则独立版本类库中的任何变化都会有导致派生类无效的潜在可能。

由于隐藏一个被继承的名字而带来的警告可以通过使用 new 修改函数加以消除。

```

class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}

```

new 修改函数显示 Derived 中的 F 是“新的”，且它确实有隐藏此继承成员的趋向。一个新成员的声明只在这个新成员的范围内隐藏一个被继承的成员。

```

class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {} // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }        // Invokes Base.F
}

```

在上面的例子中，Derived 中的 F 的声明隐藏从 Base 继承的 F，但由于 Derived 中的新 F 具有局部可访问性，它的范围并不能延伸到 MoreDerived。因此，对 MoreDerived.G 中的 F

() 的调用是有效的, 并将会产生 Base.F。

3.8 名字空间和类型名字 (Namespace And Type Names)

名字空间关键字用于声明一个范围。此命名空间范围允许你组织代码并提供了创建全局唯一类型的方法。使用名字空间指令将命名空间中包含的类型导入到最近的封闭编译单元或名字空间体中, 从而使各个类型的标识符不经限定即可使用。C#程序中的少数上下文要求 namespace-name 或 type-name 具体化。名字的每一种形式都有一个或多个被符号“.”分开的标识部分。

```
namespace-name:  
namespace-or-type-name  
  
type-name:  
namespace-or-type-name  
namespace-or-type-name:  
  
identifier  
namespace-or-type-name . identifier
```

type-name 就是作为类型的 namespace-or-type-name。在下面所述的解析中, type-name 中的 namespace-or-type-name 必须是一种类型, 否则就会出现错误。

namespace-name 就是作为名字空间的 namespace-or-type-name。在下面的解析中, namespace-name 必须是一个名字空间, 否则就会出现错误。

namespace-or-type-name 的含义确定如下:

如果 namespace-or-type-name 只有一个标识符号:

- 如果 namespace-or-type-name 出现在一个类或结构声明的主体内, 则它以那个类或结构声明开头, 后面是每一个子类或结构 (如果有), 如果有一个已知名字的成员, 这个成员是可访问的, 且表示一个类型, 则 namespace-or-type-name 指的就是那个成员。注意: 在确定 namespace-or-type-name 的含义时, 不考虑非类型成员 (构造函数、常量、域、方法、属性、索引和操作符)。否则, namespace-or-type-name 指的是从产生 namespace-or-type-name (如果有) 的名字空间声明开始, 随后是每一个子名字空间声明 (如果有), 最后到总名字空间, 然后是下列步骤, 直到实体被找到。

(1) 如果这个名字空间包括一个已知名字的名字空间成员, 则这个 namespace-or-type-name 指的就是这个成员, 并且依据这个成员而被归类为一个名字空间或类型。

(2) 否则, 如果这个名字空间声明包括一个把所给名字同一个被输入的名字空间或类型联系起来的 using-alias-directive。那么, namespace-or-type-name 指的就是那个名字空间或类型。

(3) 否则, 如果由这个名字空间声明的 using-namespace-directives 输入的名字空间只包括一种已知名字的类型, 则这个 namespace-or-type-name 指的就是那个类型。

- 如果由名字空间声明的 using-namespace-directives 输入的名字空间包括多个已知名

字的类型, 则这个 `namespace-or-type-name` 指的就是不符合实际的, 将会出现错误。

如果 `namespace-or-type-name` 有多个标识符号, `namespace-or-type-name` 的形式就是 `N.I`。在 `N.I` 中, `N` 是一个包括所有识别符号的 `namespace-or-type-name`, 但这些识别符号都不是最确切的, `I` 是最确切的识别符号。`N` 最初是被当作 `namespace-or-type-name` 而被处理的。如果对 `N` 的处理不成功, 错误出现。否则, 对 `N.I` 的解释如下:

(1) 如果 `N` 是一个名字空间, 而 `I` 是那个名字空间的可访问成员的名字, 则 `N.I` 指的就是那个成员, 并根据那个成员而被归类为一个名字空间或类型。

(2) 如果 `N` 是一个名字空间或结构类型, `I` 是 `N` 内一个可访问类型的名字, 则 `N.I` 指的就是那个类型。

(3) 否则, `N.I` 就是一个无效的 `namespace-or-type-name`, 错误出现。

全权名字 (Fully Qualified Names) 每一个名字空间或类型都有一个全权名字, 这个全权名字是唯一能从其它表示中识别出这个名字空间或类型的。一个名字空间或类型 `N` 的全权名字的确定过程如下:

- 如果 `N` 是总名字空间的一个成员, 则其全权名字就是 `N`。
- 否则, 它的全权名字就是 `S.N`。这里, `S` 是声明 `N` 的名字空间或类型的全权名字。

也就是说, `N` 的全权名字就是从总名字空间开始直至找到 `N` 的识别符号的全部路径。由于名字空间或类型的每一个成员都必须有唯一的名字, 名字空间或类型的全权名字也总是唯一的。

下面的例子就是少数带有相关全权名字的名字空间和类型的声明:

```
class A {}                // A

namespace X                // X
{
    class B                // X.B
    {
        class C {}        // X.B.C
    }

    namespace Y            // X.Y
    {
        class D {}        // X.Y.D
    }
}

namespace X.Y              // X.Y
{
    class E {}            // X.Y.E
}
```

第4章 类 型

C#语言的类型可以分为两种：值类型和引用类型。

```
type:
    value-type
    reference-type
```

类型（指针）的第三类只存在于不安全代码中。这一点将在 4.2 节进行深入探讨。值类型与引用类型的不同之处就在于值类型的变量直接包括它们的数据，而引用类型的变量则把 references 存储到它们的数据库中，后者被称为 objects。在引用类型中，两个变量可以引用同一个对象，这样，对一个变量的操作也就可能影响被另一个变量所引用的对象。在值类型中，每一个变量都有其自己的数据副本，因此，对一个变量的操作也就不可能影响另一个变量。

C#的类型系统都被这样进行统一，即一个任何类型的值都被看作一个 object。C#中的每一个类型都直接或间接地来自于 object 类型，且 object 是所有类型中最大的基类。引用类型的值都被当作对象来处理，只简单地把值看作类型 object。值类型的值通过执行封箱和非封箱操作（4.3 节）而被当作对象来处理。

4.1 值类型（Value Types）

一个值类型或者是一个结构类型，或者是一个枚举类型。C#提供了一套称为简单类型（simple types）的预先定义的结构类型。这些简单类型可以通过专用词来识别，并且可被再分为值类型、整型和浮点型。

```
value-type:
    struct-type
    enum-type

struct-type:
    type-name
    simple-type

simple-type:
    numeric-type
    bool

numeric-type:
    integral-type
    floating-point-type
    decimal

integral-type:
    sbyte
    byte
    short
```

```

ushort
int
uint
long
ulong
char
floating-point-type:
    float
    double

enum-type:
    type-name

```

所有值类型无疑都是从类 object 中继承来的。任何类型都不可能由值类型派生而来，因此，值类型是封闭式的。

一个值类型的变量总是包括那个类型的一个值。与引用类型不同，值类型的值不可能是 null，也不可能引用一个多派生类型的对象。

对一个值类型的变量进行赋值可以产生被赋值的一个复制，这与对一个引用类型的变量进行赋值不同，后者只复制引用而不复制被引用识别的对象。

4.1.1 默认构造函数 (Default Constructors)

所有值类型都自动地声明一个无参数的公共构造函数，即默认构造函数 (default constructor)。默认构造函数为值类型返回一个称为默认值 (default value) 的零初始化的实例。

- 对所有的 simple-types 来说，默认值是由所有零的一个位模式产生的。
 - sbyte、byte、short、ushort、int、uint、long、ulong 的默认值是 0。
 - char 的默认值是 '\0000'
 - float 的默认值是 0.0f
 - double 的默认值是 0.0d
 - decimal 的默认值是 0.0m
 - bool 的默认值是 false
- enum-type E 的默认值是 0
- 对于 struct-type 来说，所有值类型域都是它们的默认值，而所有的引用类型域为 null。

与其他构造函数一样，值类型的默认构造函数是通过使用 new 操作符而产生的。在下面的例子中，变量 i 和 j 都被初始化为 0。

```

class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}

```

}

由于每一个值类型都默认地带有一个无参数的公共构造函数，因此，一个结构类型不可能含有一个无参数构造函数的显式声明。然而，结构类型可以声明带有参数的构造函数。例如：

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

根据上面的声明，语句

```
point p1=new point();

point p2=new point(0,0)
```

都产生一个 x 和 y 都被初始化为 0 的 point。

4.1.2 结构类型 (Struct Types)

结构类型是一个可以声明构造函数、常量、域、方法、属性、索引、操作符和嵌套类型的值类型。结构类型将在 4.11 节中描述。

4.1.3 简单类型 (Simple Types)

C#提供了一套预先定义的结构类型称为简单类型。简单类型可以通过专用词来识别，但是，这些专用词只是 system 名字空间内预先定义的结构类型的别名。描述如表 4-1。

表 4-1 名字空间内预定义的结构类型的别名

保留字	别名
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32

(续)

保留字	别名
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

由于一个简单类型是一个结构类型的别名，每一个简单类型都有其成员。例如，int 具有在 system.int32 中声明的成员和从 system.object 中继承的成员，下列语句允许出现：

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();        // System.Int32.ToString() instance method
string t = 123.ToString();      // System.Int32.ToString() instance method
```

简单类型与其它结构类型的区别就在于它们允许一定的额外操作：

- 绝大多数的简单类型都允许通过写 literals 而产生的值（2.4.4 节）。例如，123 是类型 int 的字面值，‘a’ 是类型 char 的字面值。C# 不提供其它结构类型的字面值，其它结构类型的值最终总是通过那些结构类型的构造函数而产生。

- 当一个表达式的操作数都是简单类型常量时，编译器就可以在编译期赋值表达式。这样的表达式称为 constant-expression(7.15 节)。含有被其它结构类型定义的操作符的表达式总是提示运行时间。

- 通过鉴定 const 声明可以声明简单类型的常量（10.3 节）。但具有其它结构类型的常量时这是不可能的，但是，static readonly 域提供相似的结果。

- 含有简单类型的转换可以参加由其它结构类型定义的操作符转换的赋值，但是，一个用户自定义操作符绝不能参与另一个用户自定义操作符的赋值（6.4.2 节）。

4.1.4 整类型（Integral Types）

C# 支持 9 个整类型：sbyte、byte、short、ushort、int、uint、long、ulong 和 char。

这些整类型的值的大小和范围如下：

- sbyte 类型表示带有符号标记的值在 -128~127 之间的 8 位整数。
- byte 类型表示无正负号标记的值在 0~255 之间的 8 位整数。
- short 类型表示有正负号标记的值在 -32768~32767 之间的 16 位整数。
- ushort 类型表示无正负号标记的值在 0~65535 之间的 16 位整数。
- int 类型表示有正负号标记的值在 -2147483648~2147483647 之间的 32 位整数。

- `uint` 类型表示无正负号标记的值在 0~4292967295 之间的 32 位整数。
- `long` 类型表示有正负号标记的值在 -9223372036854775808~9223372036854775807 之间的 64 位整数。
- `ulong` 类型表示无正负号标记的值在 0~18446744073709551615 之间的 64 位整数。
- `char` 类型表示无正负号标记的值在 0~65535 之间的 16 位整数。`char` 类型的可能值的设定与单码字母相对应。

整类型的一元操作符和二进制操作符的操作总是与有符号的精确度为 32 位、64 位以及无符号精确度为 32 位和 64 位联系在一起。

- 对于一元操作符`+`和`~`来说，操作数被转化为类型 `T`，这里，`T` 是能完全表示这个操作数的可能的值 `int`、`uint`、`long`、`ulong` 的第一个字母。然后，用类型 `T` 的精确度来执行这个操作（即在这个操作中，精确度为类型 `T` 的精确度），且结果的类型为 `T`。

- 对于一元操作符`--`来说，操作数被转化为类型 `T`，`T` 是能完全表示这个操作数的可能的值 `int` 和 `long` 的第一个字母。在操作中，精确度为类型 `T` 的精确度，结果的类型为 `T`。一元操作符`--`不能用在类型 `ulong` 的操作数中。

- 对于二进制操作符`+`、`-`、`*`、`/`、`%`、`&`、`^`、`|`、`==`、`!=`、`>`、`<`、`>=`和`<=`来说，`T` 是能完全表示这个操作数的可能的值 `int`、`uint`、`long`、`ulong` 的第一个字母。在操作中，精确度为类型 `T` 的精确度，结果的类型为 `T`（对于有些操作数来说为 `bool`）。对于二进制操作符来说，一个操作数是类型 `long`，另一个是类型 `ulong` 是不可能的。

- 对于二进制操作符`<<`和`>>`来说，左边的操作符被转化为类型 `T`，`T` 是能完全表示这个操作数的可能的值 `int`、`uint`、`long`、`ulong` 的第一个字母。在操作中，精确度为类型 `T` 的精确度，结果的类型为 `T`。

类型 `char` 被归类为整类型，但它在两方面与其它整类型不同：

- 从其它类型到 `char` 型有两个固定的转换。特殊情况下，即使 `sbyte`、`byte` 和 `ushort` 类型具有用 `char` 类型能充分表示的值的范围，从 `sbyte`、`byte` 或 `ushort` 到 `char` 的固定转换也不存在。

- `char` 类型的常量必须被写为 `character-literals`。在与 `cast` 联合时，字符常量只能写为 `integer-literals`。例如，`(char) 10` 与 `"\x000a"` 的意思相同。

`checked` 和 `unchecked` 操作符用来控制整类型的算术操作和转换的检验溢出（7.5.12 节）。在 `checked` 上下文中，溢出产生一个错误的编译时间或显示 `overflowexception`。在 `unchecked` 上下文中，溢出被忽略，不适合于目的类型的高数位被删去。

4.1.5 浮点数类型 (Floating Point Types)

C#支持两个浮点数类型：`float` 和 `double`。`float` 和 `double` 分别用 32 位单精度和 64 位双精度的 IEEE754 格式来表示，它们提供了下列几套数值：

- 正零和负零。大多数情况下，正零和负零与简单的零值一致，但在某些操作中却把两者区分开来。

- 正无穷和负无穷。无穷是用一个非零数字除以零时而得到的。例如，`1.0/0.0` 的值是正无穷，`-1.0/0.0` 的值是负无穷。

- `Not-A-Number` 通常缩写为 `NaN`，`NaN` 是由无效浮点操作而产生的，如用零除以零

(0/0)。

● 式子 $s*m*2^e$ 的非零值的无穷设置。这里, s 是 1 或 -1, m 和 e 由特定的浮点类型来确定: 对于 float 来说, $0 < m < 2^{24}$ 而 $-149 \leq e \leq 104$, 对于 double 来说, $0 < m < 2^{53}$ 而 $-1075 \leq e \leq 970$ 。

float 类型可表示的值的范围大约从 $1.5*10^{-45}$ ~ $3.4*10^{38}$, 精确度为 7 位数。double 类型可表示的值的范围大约从 $-5.0*10^{-324}$ ~ $1.7*10^{308}$, 精确度为 15~16 位数。

如果有一个二进制操作符的操作数是浮点型的, 则其它的操作数必须是整类型或浮点数类型。操作过程的赋值情况如下:

● 如果有一个操作数是整类型, 那么, 这个操作数就被转换成其它操作数的浮点数类型。

● 如果两个操作数的类型都是 double, 则其它操作数也要被转化为 double, 在操作中至少要用 double 的取值范围和精确度, 且结果类型为 double (对于有些操作数来说, 也可以是 bool)。

● 否则, 操作中至少要用 float 的取值范围和精确度, 结果类型为 float (对于有些操作数来说, 也可以是 bool)。

浮点操作符, 包括赋值操作符, 决不能出现异常。在异常情况下, 浮点操作会出现零、无穷或 NaN 值, 现描述如下:

● 如果一个浮点操作的结果对于目的格式来说太大, 则操作的结果是正零或负零。

● 如果一个浮点操作的结果对于目的格式来说太小, 则操作的结果是正无穷或负无穷。

● 如果浮点操作无效, 则结果为 NaN。

● 如果浮点操作有一个或两个操作数都是 NaN, 则结果也是 NaN。

浮点操作的精确度可以比其结果的精确度高。例如, 在有些硬件结构支持一种比 double 类型取值范围和精确度更大的或长双精度的浮点数类型, 并用这种更高精确度的类型自动完成所有浮点数操作。

在执行过程中, 只有用过多的成本才能使这种结构在较低的精确度下完成浮点操作, 而且, 在 C# 中不需要取消执行和精确度, 在所有浮点数操作中都允许使用一个更高的精确度。如果不输出更精确的结果, 很难获得一个有价值的结论。然而, 在表达式 $x*y/z$ 中, 乘法运算产生的结果范围在 double 之外, 但是, 接下来的除法运算又把当前的结果带回到 double 范围之内, 表达式可被赋予更高范围的值, 这可能会导致产生一个有限的而不是无限的结果。

4.1.6 十进制类型 (The Decimal Type)

十进制(decimal)类型是一个适合于财务和金融运算的 128 位数据类型。十进制(decimal)类型可表示的值的范围大约是从 $1.0*10^{-28}$ ~ $7.9*10^{28}$ 的 28~29 位有意义的数字。类型 decimal 的值的限定公式是 $s*m*10^e$, 这里, s 的值是 1 或 -1, $0 < m < 2^{96}$, $-28 \leq e \leq 0$ 。十进制类型不支持有正负号标记的零、无穷或 NaN。

一个 decimal 可以用一个由 10 的幂度量的 96 位整型数来表示。对于绝对值小于 1.0m 的 decimals 来说, 其值可精确到十进制第 28 位, 但不能再多。对于绝对值大于或等于 1.0m 的 decimals 来说, 其值可精确到十进制第 28~29 位数字。与数据类型 float 和 double 相反, 十进制小数如 0.1 在 decimal 表达式中可确切表示。在 float 和 double 表达式中, 这样的小数通常为有限小数, 这使得那些表示更易于犯超范围的错误。

如果二进制操作符的操作数有一个是 decimal，则其它操作数也必须是整型或 decimal 类型。如果存在一个整型操作数，那么，在执行操作之前要将它转化为 decimal 型。对类型 decimal 值的操作精确到 28~29 位数字，但十进制只有 28 位。结果被四舍五入为最近的可表示值，当一个结果均匀地接近于两个可表示值时，所取的结果为这样一个值，其具有最小意义数字位上的值是那两个可表示值相应位上值的平均值。

如果十进制算术操作产生的值对于取约数后的十进制格式来说太小，则操作的结果就是零。如果十进制算术操作产生的值对这个 decimal 格式来说太小，则显示 `overflowexception`。

decimal 类型比浮点数类型更精确，但取值范围却比它小。因此，从浮点数类型到 decimal 的转换可能会出现额外溢出，而从 decimal 到浮点数类型的转换则可能会导致精度的丢失。由于这些原因，在浮点数类型和 decimal 之间没有绝对相等的转换，且即使没有明显的分类，在一个表达式中也不可能混淆浮点操作数和 decimal 操作数。

4.1.7 布尔类型 (The Bool Type)

布尔 (bool) 类型指的是布尔逻辑值。布尔类型的可能值是 true 和 false。在布尔 (bool) 和其它类型之间没有标准的转换。尤其是，布尔 (bool) 型是与整型有区别并且是独立的，布尔 (bool) 值不能用在整型值的位置，也不能用在其两侧。在 C 和 C++ 语言中，整型值零或指针 null 都可转化为布尔值 false，而一个非零整型值或一个非 null 指针则可转化为布尔型值 true。在 C# 中，这样的转换也可以通过一个整型值与零的显式对比或一个对象引用同 null 的显式对比来实现。

4.1.8 枚举类型 (Enumeration Types)

枚举类型是一个具有已命名常量的特殊类型。每一个枚举类型都有一个潜在类型，它们是 byte、sbyte、short、ushort、int、uint、long 或 ulong。枚举类型通过枚举声明来定义 (参见 14.1 节)。

4.2 引用类型 (Reference Types)

引用类型包括类类型、接口类型、数组类型或一个委托类型。

```
reference-type:
    class-type
    interface-type
    array-type
    delegate-type

class-type:
    type-name
    object
    string
```



```

interface-type:
    type-name

array-type:
    non-array-type    rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers   rank-specifier

rank-specifier:
    [ dim-separatorsopt ]

dim-separators:,
    dim-separators   ,

delegate-type:
    type-name

```

引用类型的值就是这个类型对 instance 的一个引用，后者被称为 object。特殊值 null 适用于所有引用类型，用来表示一个实例的缺省。

4.2.1 类类型 (Class Types)

类类型指的是一个数据结构，它包括数据成员（常量、域、和事件）、函数成员（方法、属性、索引、操作符、构造函数和解构函数）和嵌套类型。类类型支持继承，即派生的类可以扩展或具体化基类。类类型的实例通过使用 object-creation-expressions(参见 7.5.10.1 节)而产生。

关于类类型将在第 10 章中描述。

4.2.2 对象类型 (The Object Type)

对象 (object) 类型是所有其它类型中最基本的类。C# 中的每一个类型都直接或间接地来自于对象类型。

object 这个关键词只是预先定义的 system.object 类的一个别名。输入关键词 object 就相当于输入 system.object，反过来也是一样。

4.2.3 字符串类型 (The String Type)

字符串 (string) 类型是直接从 object 继承下来的一个封闭的类型。string 类的实例表示的是单码字符串。

字符串 (string) 类型的值可作为字符串字面值 (参见 2.4.4 节)。

关键词 string 只是预先定义的 system.string 类的一个别名。输入关键词 string 就相当于输

入 string，反过来也是一样。

4.2.4 接口类型 (Interface Types)

接口就相当于一个契约。完成接口的类或结构必须遵守这个契约。一个接口可能是从多个基本接口中继承而来的，且一个类或结构可能完成多个接口。关于接口类型将在 13 章中加以描述。

4.2.5 数组类型 (Array Type)

数组指的是通过计算索引可以访问的变量的数据结构。某一数组中所含有的变量也可以调用这个数组中类型相同的元素，且这个类型就是此数组的元素类型。

关于数组类型将在 12 章中详细介绍。

4.2.6 委托类型 (Delegate Type)

委托是一种数据结构，它引用静态方法或对象实例以及那个对象的实例方法。

在 C 或 C++ 中，委托几乎等价于函数指针。而函数指针只能引用静态函数，委托则既可以引用静态函数，也可以引用实例函数。在引用实例函数时，委托不仅可以存储对方法入口的引用，也可以存储对产生这个方法目的实例的引用。

关于委托类型将在 15 章进行描述。

4.3 封箱和非封箱 (Boxing and Unboxing)

封箱和非封箱是 C# 类型系统中的一个中心概念。它允许 value-type 的任何值和类型 object 之间的相互转换，并由此把 value-types 和 reference-types 联系起来。封箱和非封箱使得关于类型系统中任何类型的一个值在什么情况下可被看作一个对象的观点得以统一。

4.3.1 封箱转换 (Boxing Conversions)

封箱转换允许任何一个 value-type 被等值地转化为类型 object 或由 value-type 执行的任何一个 interface-type。封箱 value-type 的作用包括给实例分配对象实例以及复制 value-type 的值。

封箱 value-type 的取值过程的最好解释就是想象存在那个类型的一个 boxing class。对于任何 value-type T，封箱类声明如下：

```
class T_Box
{
    T value;
```

```

T_Box(T t) {
    value = t;
}

```

这里, 类型 `T` 的一个值 `v` 的封箱包括: 执行表达式 `new T_Box(v)` 及将结果实例以类型 `object` 值的形式返回。因此, 语句

```
int i=123; object box=i;
```

在概念上与下面的语句一致:

```
int i=123; object box=new int_Box (i);
```

上面的封箱类如 `T_Box` 和 `int_Box` 实际上并不存在, 且一个封箱值的动态类型实际上并不是一个类类型, 类型 `T` 的封箱值具有动态类型 `T`, 且如果用 `is` 操作符的动态类型来检验, 只能引用类型 `T`。例如:

```

int i = 123;
object box = i;
if (box is int) {
    Console.WriteLine("Box contains an int");
}

```

此程序将会把字符串 “box contains an int” 输出到控制函数上。

封箱转换的涵义就是对被封箱的值做一个复制。这与从 `reference-type` 到类型 `object` 的转换不同, 在后者中, 这个值继续引用同样的实例, 且只是简单地被看作为充分派生的类型 `object`。例如, 假设声明如下:

```

struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

下面的语句:

```

Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.WriteLine(((Point)box).x);

```

将会在控制函数上输出值 10, 因为在 `p` 给 `box` 赋值时进行的隐式封箱操作使得 `p` 的值被

复制。如果 point 被声明为一个 class，则输出值为 20，因为 p 和 box 引用同样的实例。

4.3.2 非封箱转换 (Unboxing Conversions)

非封箱转换允许从类型 object 到任何一个 value-type 或从任何一个 interface-type 到执行 interface-type 的任何一个 value-type 的显式转换。非封箱操作包括以下几步：首先，确定对象实例是所给 value-type 的一个被封箱的值，其次，在异常时复制这个值。

根据前面所提到的假想的封箱类，从对象 box 到 value-type T 的非封箱转换包括执行表达式((T_box)box).value。因此，语句：

```
object box=123;

int i=(int) box;
```

从概念上与下面的语句一致：

```
object box=new int_box (123);
int i=((int_box) box).value;
```

要使所给 value-type 的非封箱转换在运行期内成功完成，原自变量的值必须是对先前封箱那个 value-type 的值时所产生的一个对象的引用。如果这个原自变量是 null 或对不相融对象的一个引用，则将会显示 InvalidCastException。

第5章 变 量

变量就是存储地址。每一个变量都有一个类型，它决定什么样的值可以存储在这个变量内。C#是一种类型安全语言，C#编译函数能保证存储在变量里的值的类型总是正确的。一个变量内的值可以通过赋值或操作符++和--的使用而改变。

一个变量在它获得值以前必须被 **definitely assigned**（参见 5.3 节）。

如下所述，变量要么是 **initially assigned**，要么是 **initially unassigned**。一个初始化赋值变量有定义好的初始值，且总被认为是被明确赋值的。一个初始化未赋值变量无初始值。对于一个在特定地址中被认为明确赋值的初始化未赋值变量来说，对它的赋值必须发生在通往其地址的每一个可能的路径。

5.1 变量分类 (Variable Categories)

C#中有 7 类变量：静态变量、实例变量、数组成员、值参数、引用参数、输出变量和局部变量。以下几部分将对它们进行逐一描述。

在下面的例子中：

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

x 是一个静态变量，y 是一个实例变量，v[0]是一个数组成员，a 是一个值参数，b 是一个引用参数，c 是一个输出参数，而 i 是一个局部变量。

5.1.1 静态变量 (Static Variables)

声明时带有 **static** 修改函数的域称为 **static variable**。当静态变量所声明的类型被装载时（参见 10.11 节），它才得以存在。

一个静态变量的初始值就是这个变量类型的默认值（参见 5.2 节）。为便于明确赋值检验，通常认为静态变量是被初始化赋值的。

5.1.2 实例变量 (Instance Variables)

声明时没有 `static` 修改函数的域称为 instance variables。

5.1.2.1 类中的实例变量 (Instance Variables In Classes)

当一个类产生一个新的实例时，这个类的实例变量也开始出现。当这个实例的析构函数（如果有）已经执行，不再引用这个实例时，它也就停止存在。

一个类的实例变量的初始值就是这个变量类型的默认值（参见 5.2 节）。

为便于明确赋值检验，通常认为一个类的实例变量是被初始化赋值的。

5.1.2.2 结构中的实例变量 (Instance Variables In Structs)

结构中的实例变量与它所属的结构变量具有相同的寿命。也就是说，当一个结构类型的变量开始出现或消失时，这个结构的实例变量随之出现或消失。

结构中的实例变量的初始化赋值状态与包含它的结构变量的初始化赋值状态一致。也就是说，当一个结构变量被初始化赋值时，其实例变量也被初始化赋值；当一个结构变量未被初始化赋值时，其实例变量也未被初始化赋值。

5.1.3 数组成员 (Array Elements)

当一个数组实例产生时，这个数组的成员也就产生了，当那个数组实例不再被引用时，其数组成员也就消失了。

每一个数组成员的初始值都是这个数组成员类型的默认值（参见 5.2 节）。

为便于明确赋值检验，通常认为数组元素是被初始化赋值的。

5.1.4 值参数 (Value Parameters)

声明时没有 `ref` 或 `out` 修改函数的参数称为 value parameters。

值参数是在它所属的函数成员（方法、构造函数、访问函数、操作符）引用的基础上产生的，它被初始化为引用中所给的自变量的值。当这个函数成员返回时，值参数也就消失。

为便于明确赋值检验，通常认为值参数是被初始化赋值的。

5.1.5 引用参数 (Reference Parameters)

声明时带有 `ref` 修改函数的参数就是一个 reference parameter。

引用参数不产生新的存储地址。而是与函数成员引用时作为自变量的变量具有相同的存储空间。因此，引用参数的值总是与所隐藏的变量的值相同。

下面的明确赋值规则适用于引用参数。注意，它们与 5.1.6 中所述的输出参数的规则不同。

- 一个变量在其作为函数成员引用中的引用参数而被使用之前必须被明确赋值。
- 在函数成员内，通常认为引用参数是被初始化赋值的。

在一个结构类型的实例方法或实例访问函数内，关键词 `this` 相当于这个结构类型（参见 7.5.7 节）的一个引用参数。

5.1.6 输出参数（Output Parameters）

声明时带有 `out` 修改函数的参数就是一个 `output parameter`。

输出参数不产生新的存储空间。而是与函数成员引用时作为自变量的变量具有相同的存储空间。因此，输出参数的值总是与所隐藏变量的值相同。

下面的明确赋值规则适用于输出参数。注意，它们与 5.1.5 节中所述的引用参数的规则不同。

- 一个变量在其作为函数成员引用中的输出参数而被使用之前必须被明确赋值。
- 在函数成员引用过程中，认为作为输出参数的每一个变量在执行路径中都是被赋值的。
- 在一个函数成员内，认为输出参数是未初始化赋值的。
- 一个函数成员的每一个输出参数在函数成员返回之前必须被明确赋值（参见 5.3 节）。

在一个结构类型的构造函数内，关键词 `this` 相当于这个结构类型（参见 7.5.7）的一个输出参数。

5.1.7 局部变量（Local Variables）

局部变量（`local variable`）是由 `local-variable-declaration` 声明的。这可能会发生在 `block`、`for-statement`、`switch-statement` 或 `using-statement` 之内。当控制进入直接包括局部变量声明的 `block`、`for-statement`、`switch-statement` 或 `using-statement` 时就产生一个局部变量。当控制离开它的 `block`、`for-statement`、`switch-statement` 或 `using-statement` 时，局部变量就消失。局部变量不是自动初始化的，因此，也就没有默认值。为便于明确赋值检验，通常认为局部变量是未被初始化赋值的。`local-variable-declaration` 可能包括 `variable-initializer`，在这种情况下，除了在 `variable-initializer` 所提供的表达式内外，这个变量都被认为是被明确赋值的。

在一个局部变量范围内，在 `variable-declaration` 之前的文本位置引用局部变量是错误的。

注意：这里的“开始存在”和“停止存在”指的是范围，而不是无用单元收集程序，这一点我们必须弄清。在描述此语言的过程中，应该有充分的余地使贪婪的执行过程能够识别不能再用的局部变量以及不再被引用的对象，以便使它决定把这些被引用过的对象放入无用单元收集程序。上面的句法可能会导致对象在局部变量越过其范围或“停止存在”之前被无用单元收集程序收集。

5.2 默认值（Default Values）

以下几类变量是自动初始化为它们的默认值的：

- 静态变量。

- 类实例的实例变量。
- 数组成员。

一个变量的默认值取决于这个变量的类型，其确定如下：

- 对于 value-type 的一个变量，其默认值与由 value-type 的默认构造函数（参见 4.1.1 节）所算出的值相同。
- 对于 reference-type 的一个变量来说，其默认值是 null。

5.3 明确赋值 (Definite Assignment)

在一个函数成员的可执行代码的给定空间内，如果编译函数可以通过静态流动分析证明一个变量已被自动初始化或已成为至少一个赋值的目标，那么，就说这个变量是 **definitely assigned**。明确赋值的规则是：

- 一个初始化赋值变量（参见 5.3.1 节）总被认为是被明确赋值的。
- 一个初始化未赋值变量（参见 5.3.2 节）在特定位置是被明确赋值的，那就是如果通往这个位置的所有可能的执行路径至少包括下列几项中的一项时：
 - 这个变量是左操作数的一个简单赋值（参见 7.13.1 节）。
 - 一个引用表达式（参见 7.5.5 节）或将这个变量变成一个输出参数的对象产生表达式（参见 7.5.10.1 节）。
 - 对于一个局部变量来说，含有变量初始化函数的一个局部变量声明（参见 8.5 节）。

一个 struct-type 变量的实例变量的明确赋值状态既可以集中探索又可以分别研究。除上面的规则外，下面的规则也适用于 struct-type 变量及其实例变量：

- 如果一个实例变量所属的 struct-type 变量是被明确赋值的，则这个实例变量也是被明确赋值的。
- 如果 struct-type 变量的每一个实例变量都是被明确赋值的，则这个 struct-type 也是被明确赋值的。

在下列上下文中要求明确赋值：

- 一个变量在其值可被得到的每一个空间都必须被明确赋值。这样，能保证从不会出现未定义的值。除下列情况外，在表达式中出现的变量都是为了取得这个变量的值：
 - 这个变量是一个简单赋值的左操作数
 - 这个变量被当作一个输出参数而使用
 - 这个变量是一个 struct-type 变量且作为一个成员访问的左操作数
- 一个变量在其被当作一个引用参数而使用的每一个空间都必须被明确赋值。这样，能保证正在引用的函数成员认为这个引用参数是初始化赋值的。
- 一个函数成员的所有输出参数在这个函数成员返回（通过一个 return 语句或通过一个到达这个函数成员主体的终点的操作）的每一个位置都必须被明确赋值。这能保证函数成员在输出参数中不返回一个未定义的值，这样，就使得编译函数认为把一个变量看作一个输出参数的函数成员引用就相当于对这个变量进行赋值。
- 一个 struct-type 构造函数的 this 变量在这个构造函数返回的每一个位置都必须被明确赋值。

下面的例子表明一个 try 语句的不同的块如何影响明确赋值:

```
class A
{
    static void F() {
        int i, j;
        try {
            // neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
            j = 2;
            // i and j definitely assigned
        }
        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }
        finally {
            // neither i nor j definitely assigned
            i = 4;
            // i definitely assigned
            j = 5;
            // i and j definitely assigned
        }
        // i and j definitely assigned
    }
}
```

确定一个变量的明确赋值状态的静态流动分析考虑到操作符&&、||、和 ? 的特殊用法。在这个例子的每一个方法中变量 i 在 if 语句的嵌套语句中都是被明确赋值的,但在其它语句中不是。

```
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
        // i not definitely assigned
    }

    static void G(int x, int y) {
```

```

int i;
if (x >= 0 || (i = y) >= 0) {           // i not definitely assigned
}
else {                                   // i definitely assigned
}                                       // i not definitely assigned
}

```

在 F 方法的 if 语句中，变量 i 在第一个嵌套语句中被明确赋值，因为表达式 (i=y) 的执行总是在这个嵌套语句执行之前。而变量 i 在第二个嵌套语句中未被明确赋值，因为变量 i 可能是未被赋值的。特殊情况下，如果变量 x 的值是负，则变量 i 是未被赋值的。同样，在方法 G 中，变量 i 在第二个嵌套语句中是被明确赋值的，但在第一个嵌套语句中却不是。

5.3.1 初始赋值变量 (Initially Assigned Variables)

下面几类变量被认为是被初始赋值的：

- 静态变量。
- 类实例的实例变量。
- 初始赋值结构变量的实例变量。
- 数组成员。
- 值参数。
- 引用参数。

5.3.2 初始未赋值变量 (Initially Unassigned Variables)

下面几类变量是初始未赋值的：

- 初始未赋值结构变量的实例变量。
- 包括结构构造函数的 this 变量在内的输出参数。
- 局部变量。

5.4 变量引用 (Variable References)

variable-reference 被看作是变量的一个 expression。一个 variable-reference 指的是一个存储空间，它在调取当前值或存储一个新值时均可被访问。在 C 和 C++ 中，一个 variable-reference 就是一个 value。

```

variable-reference:
expression

```

在下面的结构中要求一个 expression 就是一个 variable-reference:

- 一个 assignment (也可能是一个属性访问或一个索引访问) 的左边。
- 在方法或构造函数引用中被当作 ref 或 out 参数使用的自变量。

第6章 转 换

6.1 隐式转换 (Implicit Conversions)

下面的转换在分类学上称为隐式转换:

- 一致性转换。
- 隐式数值转换。
- 隐式枚举转换。
- 隐式参照转换。
- 封箱转换。
- 隐式常量表达式转换。
- 用户自定义隐式转换。

隐式转换在很多情况下都可以发生, 包括函数成员的引用过程 (参见 7.4.3 节)、cast 表达式 (参见 7.6.8 节) 以及赋值过程中 (参见 7.13 节)。

预先定义的隐式转换总能成功, 从未出现过异常。精心设计的用户自定义隐式转换也应具有这些特征。

6.1.1 一致性转换 (Identity Conversion)

一致性转换指的是任何类型之间的转换。这种转换是这样存在的, 即具有所要求类型的实体之间可以互相转换。

6.1.2 隐式数值转换 (Implicit Numeric Conversions)

隐式数值转换指的是:

- 从 byte 到 short、int、long、float、double 或 decimal 的转换。
- 从 byte 到 short、ushort、int、uint、long、ulong、float、double 或 decimal 的转换。
- 从 short 到 int、long、float、double 或 decimal 的转换。
- 从 ushort 到 int、uint、long、ulong、float、double 或 decimal 的转换。
- 从 int 到 long、float、double 或 decimal 的转换。
- 从 uint 到 long、ulong、float、double 或 decimal 的转换。
- 从 long 到 float、double 或 decimal 的转换。
- 从 ulong 到 float、double 或 decimal 的转换。
- 从 char 到 ushort、int、uint、long、ulong、float、double 或 decimal 的转换。
- 从 float 到 double 的转换。
- 从 int、uint 或 long 到 float 以及从 long 到 double 的转变可能会导致精确性的丢失,

但决不会导致其量值的丢失。其它的隐式数值转换不丢失任何信息。不存在到类型 `char` 的隐式转换。

6.1.3 隐式枚举转换 (Implicit Enumeration Conversions)

隐式枚举转换允许 `decimal-integer-literals 0` 转换为任何 `enum-type`。

6.1.4 隐式参照转换 (Implicit Reference Conversions)

隐式参照转换是指：

- 从任何 `reference-type` 到 `object` 的转换。
- 从任何 `class-type S` 到任何 `class-type T` 的转换，假设 `S` 是从 `T` 派生而来的。
- 从任何 `class-type S` 到任何 `interface-type T` 的转换，假设 `S` 是 `T` 的补充。
- 从任何 `interface-type S` 到任何 `interface-type T` 的转换，假设 `S` 是从 `T` 派生而来的。
- 从一个具有成员类型 `SE` 的 `array-type S` 到一个具有成员类型 `TE` 的 `array-type T` 的转换，假设下列条件都符合：
 - `S` 和 `T` 只有元素类型不同。也就是说，`s` 和 `t` 的大小相同。
 - `SE` 和 `TE` 都是 `reference-type`。
 - 存在从 `SE` 到 `TE` 的隐式参照转换。
- 从任何 `array-type` 到 `System.Array` 的转换。
- 从任何 `delegate-type` 到 `System.Delegate` 的转换。
- 从任何 `array-type` 或 `delegate-type` 到 `System.ICloneable` 的转换。
- 从 `null` 类型到任何 `reference-type` 的转换。

隐式参照转换就是那些在 `reference-type` 之间的转换，这些转换证明总是成功的，因此，运行其间不需要检查。

不管是隐式还是显式参照转换，都不会改变被转换对象作为参考的一致性。换句话说，虽然一个参照转换可能会改变一个值的类型，但它绝不会改变这个值本身。

6.1.5 封箱转换 (Boxing Conversions)

封箱转换允许任何 `value-type` 被隐式转换为类型 `object` 或任何由 `value-type` 执行的 `interface-type`。封箱 `value-type` 的一个值包括分配一个对象实例到那个实例中以及为那个实例复制这个 `value-type` 的值。

关于封箱转换，在 4.3.1 节中有更进一步的描述。

6.1.6 隐式常量表达式转换 (Implicit Constant Expression Conversions)

隐式常量表达式转换允许下面的转换：

- 类型 `int` 的一个 `constant-expression` (参见 7.15) 可被转化为类型 `sbyte`、`byte`、`short`、`ushort`、`uint`、`ubory`，假设这个 `constant-expression` 的值在目标类型的范围内。

- 类型 `long` 的一个 `constant-expression` 可被转化为类型 `ulong`，假设这个 `constant-expression` 的值不是负值。

6.1.7 用户自定义隐式转换 (User-defined Implicit Conversions)

用户自定义隐式转换包括一个可选择的标准隐式转换，然后是由用户自定义隐式转换操作符的执行，再往后是另一个可选择的标准隐式转换。给用户自定义转换赋值的规则参见 6.4.3 节。

6.2 显式转换 (Explicit Conversions)

下面的转换在分类上称为显式转换：

- 所有的隐式转换。
- 显式数值转换。
- 显式枚举转换。
- 显式参照转换。
- 显式接口转换。
- 非封箱转换。
- 用户自定义显式转换。

显式转换可能发生在 `cast` 表达式中（参见 7.6.8 节）。

显式转换并非总是成功，可能丢失信息，这种转换也存在于那些明显不同而不能进行显式标记的类型领域。

固定的显式转换包括所有的隐式转换。其特殊意义就在于允许丰富的 `cast` 表达式。

6.2.1 显式数值转换 (Explicit Numeric Conversions)

显式数值转换是从一个 `numeric-type` 到另一个 `numeric-type` 的转换，正因为这个原因，显式数值转换（参见 6.1.2 节）并非已经存在。

- 从 `sbyte` 到 `byte`、`ushort`、`uint`、`ulong` 或 `char` 的转换。
- 从 `byte` 到 `sbyte` 和 `char` 的转换。
- 从 `short` 到 `sbyte`、`byte`、`ushort`、`uint`、`ulong` 或 `char` 的转换。
- 从 `ushort` 到 `sbyte`、`byte`、`short` 或 `char` 的转换。
- 从 `int` 到 `sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong` 或 `char` 的转换。
- 从 `uint` 到 `sbyte`、`byte`、`short`、`ushort`、`int` 或 `char` 的转换。
- 从 `long` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`ulong` 或 `char` 的转换。
- 从 `ulong` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long` 或 `char` 的转换。
- 从 `char` 到 `sbyte`、`byte` 或 `short` 的转换。
- 从 `float` 到 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char` 或 `decimal` 的转换。

- 从 double 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 decimal 的转换。
- 从 decimal 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float 或 double 的转换。

由于显式转换包括所有的显式和隐式数值转换，用 cast 表达式（参见 7.6.8 节）实现从任何 numeric-type 到任何其它的 numeric-type 的转换总是可能的。

显式数值转换可能丢失信息，也可能出现异常情况。其过程如下：

- 对于从一整类型到另一整类型的转换来说，这个过程取决于转换发生的上下文的检验溢出（参见 7.5.12）。
- 在 checked 上下文中，如果原自变量在目的类型的范围中，则转换成功，如果不在则显示 OverflowException。
- 在 unchecked 上下文中，转换总是成功的，且只包括舍弃此原值最有意义的值这样一个步骤。
- 对于从 float、double 或 decimal 到整类型的转换来说，原值约等于 0 这个最近的整型值，且这个整型值就是转换的结果。如果结果的整型值在目标类型的范围之外，则显示 OverflowException。
- 对于一个从 double 到 float 的转换来说，double 值约等于最近的 float 值。如果 double 值太小而不能表示 float，则结果为零或负零。如果 double 值太大而不能表示 float，则结果为正无穷或负无穷。如果 double 值是 NaN，则结果也是 NaN。
- 对于一个从 float 或 double 到 decimal 的转换来说，原值转化为用 decimal 表示的值，且如果需要的话（参见 4.1.6 节），保留到第 28 位十进制位数之后最近的数字。如果原值太小而不能表示 decimal，则结果为零。如果原值是 NaN、无穷或太大，则显示 InvalidCastException。
- 对于一个从 decimal 到 float 或 double 的转换来说，decimal 值约等于最接近的 double 或 float 值。尽管这种转换可能会造成精确性的丢失，但绝不会出现异常。

6.2.2 显式枚举转换（Explicit Enum Conversions）

显式枚举转换是指：

- 从 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 decimal 到任何 enum-type 的转换。
- 从任何 enum-type 到 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double 或 decimal 的转换。
- 从任何 enum-type 到任何其它 enum-type 的转换。

两个类型之间的显式枚举转换过程就是，把任何一个参与 enum-type 的都看作那个 enum-type 的潜在类型，然后再进行这两个结果类型之间的显式或隐式数值转换。例如，给定一个具有或潜在具有类型 int 的一个 enum-type E，那么，从 E 到 Byte 到的转换过程就是一个从 int 到 byte 的显式数值转换（参见 6.2.1 节），而从 Byte 到 E 的转换则是一个从 Byte 到 int 的隐式数值转换（参见 6.1.2 节）。

6.2.3 显式引用转换 (Explicit Reference Conversions)

显式引用转换是指：

- 从 object 到任何 reference-type 的转换。
- 从任何 class-type S 到任何 class-type T 的转换，假设 S 是 T 的一个基本类型。
- 从任何 class-type S 到任何 interface-type T 的转换，假设 S 不是密封的且 S 不执行 T。
- 从任何 interface-type S 到任何 class-type T 的转换，假设 T 不是密封的，且 T 执行 S。
- 从任何 interface-type S 到任何 interface-type T 的转换，假设 S 不是从 T 派生过来的。
- 从一个带有元素类型 S_E 的 array-type S 到一个带有元素类型 T_E 的 array-type T 的转换，假设下列条件都符合：

- S 和 T 只有元素类型不同。即 S 和 T 的大小相等。
- S_E 和 T_E 都是 reference-types。
- 存在从 S_E 到 T_E 的显式引用转换。

- 从 System.Array 到任何 array-type 的转换。
- 从 System.Delegate 到任何 delegate-type 的转换。
- 从 System.ICloneable 到任何 array-type 或 delegate-type 的转换。

显式引用转换就是那些在 reference-types 之间的转换，运行期内需要对他们进行检查以确保转换正确。

要使显式引用转换在运行期内成功运行，原变量的值必须是 null，或者它所引用的对象的 actual 类型可被某一隐式引用转换（参见 6.1.4 节）转化为目标类型。如果一个显式引用转换失败了，则显式无效的转换表达式。

无论隐式还是显式引用转换，都不能改变正转化对象引用的一致性。也就是说，尽管一个引用转换可能会改变一个值的类型，但它绝不会改变这个值本身。

6.2.4 非封箱转换 (Unboxing Conversions)

非封箱转换允许从类型 object 到任何 value-type 或从任何 interface-type 到任何执行 interface-type 的 value-type 的显式转换。在非封箱操作中，首先要确保对象实例是所给 value-type 的一个被封箱的值，然后在这个实例上复制这个值。

在 4.3.2 节中对非封箱转换进行了更详细的描述。关键词 String 只是预先定义的 System.String 类的一个别名。输入关键词 String 就相当于输入 System.String，反过来也是一样。

6.2.5 用户自定义显式转换 (User-Defined Explicit Conversions)

用户自定义显式转换包括一个可选择的标准显式转换，然后是一个用户自定义隐式或显式操作符的执行，然后是另一个可选择的标准显式转换。用户自定义转换确切的赋值规则将在参见 6.4.4 节中进行描述。

6.3 标准转换 (Standard Conversions)

标准转换就是那些预先定义的转换，这些转换可作为用户自定义转换的一部分而发生。

6.3.1 标准隐式转换 (Standard Implicit Conversions)

下面的隐式转换就是标准隐式转换：

- 一致性转换 (参见 6.1.1 节)
- 隐式数值转换 (参见 6.1.2 节)
- 隐式引用转换 (参见 6.1.4 节)
- 封箱转换 (参见 6.1.5 节)
- 隐式常量表达式转换 (参见 6.1.6 节)

标准隐式转换不包括用户自定义隐式转换。

6.3.2 标准显式转换 (Standard Explicit Conversions)

标准显式转换是指所有的标准隐式转换加上显式转换中与标准隐式转换对应的子转换。也就是说，如果从类型 A 到类型 B 存在一个标准隐式转换，那么，从类型 A 到类型 B 以及从类型 B 到类型 A 均存在一个标准显式转换。

6.4 用户自定义转换 (User-Defined Conversions)

C# 允许 `user-defined conversions` 扩大预先定义的隐式或显式转换。用户自定义转换是通过在类或结构类型中声明转换符号 (参见 10.9.3 节) 而被引入的。

6.4.1 被允许的用户自定义转换 (Permitted User-Defined Conversions)

C# 只允许声明特定的用户自定义转换。尤其是不能重新定义一个已经存在的隐式或显式转换。只有下列条件都符合时，才允许类或结构声明一个从原类型 S 到目标类型 T 的转化：

- S 和 T 的类型不同。
- S 和 T 中有一个是声明这个操作符的类或结构类型。
- S 和 T 都不是 `object` 或 `interface-type`。
- T 不是 S 的一个基类，S 也不是 T 的一个基类。

适用于用户自定义转换的限制性条件将在参见 10.9.3 节中进行深入探讨。

6.4.2 用户自定义转换求值 (Evaluation of User-Defined Conversions)

用户自定义转换求值把一个值从其类型 (即 `source type`) 转化为另一种类型，即 `Target type`。用户自定义转换求值的最终目标就是要为特定的原类型和目标类型找到 `The Most Specific` 用户自定义转换操作符。这个过程可以分解为以下步骤：

(1) 找到用户自定义转换操作符所出自的类或结构的集合。这个集合包括原类型及其基类和目标类型及其基类（假设只有类或结构可以声明用户自定义操作符，且非类类型无基类）。

(2) 从那个类型集合中确定哪些用户自定义转换操作符是可用的。一个可用的转换操作符必须能够完成从原类型到这个操作符的自变量类型的标准转换（参见 6.3 节），且它必须能够完成从这个操作符的结果类型到目标类型的标准转换。

(3) 从可用的用户自定义操作符集合中，确定哪一个操作符明显是最专化的。一般来说，最专化的操作符就是那种自变量类型与原类型最接近而结果类型与目标类型最接近的操作符。确定最专化用户自定义转换操作符的确切规则将在下面几部分加以说明。

一旦最专化的用户自定义转换操作符被确认，用户自定义转换的实际操作可分为以下三步：

(1) 首先，如果需要的话，执行一个从原类型到用户自定义转换操作符自变量类型的标准转换。

(2) 其次，引用用户自定义转换操作符完成转换。

(3) 最后，如果需要的话，执行一个从用户自定义转换操作符的结果类型到目标类型的标准转换。

用户自定义转换求值所包括的用户自定义转换操作符从不超过一个。也就是说，从类型 S 到类型 T 的转换绝不会首先执行一个从 S 到 X 的用户自定义转换，再执行一个从 X 到 T 的用户自定义转换。

6.4.3 用户自定义隐式转换 (User-Defined Implicit Conversions)

从类型 S 到类型 T 的用户自定义隐式转化过程如下：

- 找到用户自定义转换操作符所在的类型集合 D。这个集合包括 S（如果 S 是一个类或结构）、S 的基类（如果 S 是一个类）、T（如果 T 是一个类或结构）以及 T 的基类（如果 T 是一个类）。
- 找到可用的用户自定义转换操作符的集合 U。这个集合包括被 D 中的一个类或结构声明的用户自定义隐式转换操作符。这些操作符能实现从包含 S 的类型到 T 包含的类型之间的转化。

如果 U 是空的，则转换就是未定义的，将会出现错误。

- 在 U 中找到这些操作符的最专化原类型 S_x；
- 如果 U 中的任何操作符都是从 S 转化的，那么 S_x 就是 S。
- 否则，S_x 就是 U 中这些操作符原类型的联合集合所包括的类型。如果找不到包含它的类型，那么这个转换就是不明确的，将会出现错误。
- 找到 U 中这些操作符的最专化目标类型 T_x：
 - 如果 U 中的任何操作符都转化为 T，则 T_x 就是 T。
 - 否则，T_x 就是 U 中这些操作符的目标类型的联合集合中容量最大的类型。如果找不到容量最大的类型，则这个转换就是模糊的，将会出现错误。
- 如果 U 确实只包含从 S_x 转化为 T_x 的一个用户自定义转换操作符，那么，这就是最

专化的转换操作符。如果不存在或存在多个这样的操作符，那么，这个转换就是不清楚的、错误的。否则，用户自定义转换的应用如下：

- 如果 S 不是 Sx，则从 S 到 Sx 将执行一个标准隐式转换。
- 在从 Sx 转化为 Tx 时，用最专化的用户自定义转换操作符。
- 如果 Tx 不是 T，则执行一个从 Tx 到 T 的标准隐式转换。

6.4.4 用户自定义显式转换 (User-Defined Explicit Conversions)

从类型 S 到类型 T 的用户自定义显式转换过程如下：

- 找到用户自定义转换操作符所在类型的集合 D。这个集合包括 S（如果 S 是一个类或结构）、S 的基类（如果 S 是一个类）、T（如果 T 是一个类或结构）以及 T 的基类（如果 T 是一类类）。
- 找到可用的用户自定义转换操作符的集合 U。这个集合包括被 D 中的某些类或结构声明的用户自定义隐式或显式转换操作符，这些操作符能完成从包含 S 或被 S 包含的类型到包含 T 或被 T 包含的类型的转化。如果 U 是空的，则这个转换就是未定义的，将会出现错误。
- 在 U 中找到这些操作符的最专化原类型 Sx；
 - 如果 U 中的每个操作符都是由 S 转化而来的，则 Sx 就是 S。
 - 否则，如果 U 中的任何操作符都是从包含 S 的类型转化而来的，那么，Sx 就是那些操作符的原类型的联合集合所包括的类型。如果找不到所包括的类型，则这个转换就是不清楚的、错误的。
 - 否则，Sx 就是 U 中这些操作符的原始类型的联合集合中容量最大的类型。如果找不到容量最大的类型，则这个转换就是不清楚的，将会出现错误。
- 找到 U 中这些操作符的最专化目标类型 Tx；
 - 如果 U 中的任何操作符都转化为 T，则 Tx 就是 T。
 - 否则，如果 U 中的任何操作符都转化为被 T 所包括的类型，则 Tx 就是那些操作符原类型的联合集合中容量最大的类型。如果找不到容量最大的类型，则转换就是不清楚的，错误的。
 - 否则，Tx 就那些操作符目标类型的联合集合所包括的类型。如果找不到所包括的类型，则转换就是不清楚的，错误的。
- 如果 U 只包括一个从 Sx 转化为 Tx 的用户自定义转换操作符，则这个操作符就是最专化转换操作符。
- 如果不存在或存在多个这样的操作符，则转换就是不清楚的、错误的。否则，用户自定义转换被应用如下：
 - 如果 S 不是 Sx，则执行一个从 S 到 Sx 标准显式转换。
 - 从 S 转化为 Tx，用最专化用户自定义转换操作符。
 - 如果 Tx 不是 T，则执行一个从 Tx 到 T 标准显式转换。

第7章 表达式

表达式由一系列操作符以及用来说明算法的操作数组成。本章将对句法、求值顺序以及表达式的意义进行说明。

7.1 表达式分类 (Expression Classifications)

7.1.1 分类 (Classifications)

表达式可以被分为以下几类：

- 值，每一个值都有一个相关的类型。
- 变量，每一个变量都有一个相关的类型，即这个变量所声明的类型。
- 名字空间，这种表达式只能出现在 `member-access` 的左边。在任何其它上下文中，名字空间表达式都会导致错误出现。
- 类型，这种表达式只能出现在 `member-access` 的左边或作为操作符 `as`、`is` 或 `type of` 的操作数。在任何其它上下文中，类型表达式都会导致错误出现。
- 方法群，即成员检索时而产生的一系列重载方法，一个方法群可以具有相关的实例表达式。当一个实例方法被运用时，对该实例表达式求值的结果就是由 `this` 所表示的实例。方法群只允许出现在 `invocation-expression` 或 `delegate-creation-expression` 中。在任何其它上下文中，方法群表达式都会导致错误出现。
- 属性访问，每一个属性访问都有一个相关的类型，称为这个属性的类型。而且属性访问还可以带有相关的实例表达式。当引用实例属性访问的访问函数 (`get` 或 `set` 块) 时，对该实例表达式进行求值的结果就是由 `this` 表示的实例。
- 事件访问，每一个事件访问都有一个相关的类型，称为这个事件的类型。事件访问也可以带有相关的实例表达式。事件访问可以作为操作符 `+=` 和 `-=` 的左操作数而出现。在任何其它上下文中，事件访问表达式都将导致错误出现。
- 索引访问，每一个索引访问都有一个相关的类型，称为这个索引的元素类型。索引访问可以带有相关的实例表达式及相关的自变量列表。当索引访问的访问函数 (`get` 或 `set` 块) 被引用时，对其实例表达式求值的结果就是由 `this` 表示的实例，且对其自变量列表求值的结果就是此引用的参数列表。
- 空白，发生在当表达式是对一返回类型为 `void` 的方法的引用时。空表达式只在 `statement-expression` 的上下文中有效。

表达式的最后结果不能是名字空间、类型、方法群或事件访问。而且，如上所述，表达式的这些种类只能通过引用 `get-accessor` 或 `set-accessor` 而产生其值，特殊的访问由属性或索引访问的上下文确定；如果访问是一赋值的目标，那么，`set-accessor` 的作用就是赋一个新值。

否则, get-accessor 的作用就是取得当前值。

7.1.2 表达式的值 (Values of Expressions)

绝大多数含有表达式的结构, 都需要表达式表示一个 value。在这种情况下, 如果实际表达式表示的是名字空间、类型、方法群或空白, 则将会出现错误。然而, 如果表达式表示的是属性访问、索引访问或变量, 那么这个属性、索引或变量的值就被自动取代:

- 变量的值指的是当前存储在由这个变量识别的存储空间里的值, 一般认为一个变量在取得其值之前是被明确赋值的, 否则, 编译时将会出现错误。
- 属性访问表达式的值是通过引用该属性的 get-accessor 而取得的, 如果一个属性没有 get-accessor, 那么, 将会出现错误。否则, 如果有将执行一个函数成员引用, 且引用的结果就是这个属性访问表达式的值。
- 索引访问表达式的值是通过引用该索引的 get-accessor 而获得的。如果一个索引没有 get-accessor, 则将会出现错误。否则, 将执行一个函数成员引用, 这个函数成员引用带有一个与此索引访问表达式有关的自变量列表, 且引用的结果就是这个索引访问表达式的值。

7.2 操作符 (Operators)

表达式是由 operands 和 operators 构成的, 一个表达式的操作符表明要对其操作数进行哪些操作。操作符包括 +、-、*、/ 和 new, 操作数包括字母、域、局部变量和表达式。

操作符有三种类型:

- 一元操作符, 一元操作符只有一个操作数且使用前缀符号 (如 -x) 或后缀符号 (如 x++)。
- 二元操作符, 二元操作符有两个操作数且都要使用中缀符号 (如 x+y)。
- 三元操作符, 三元操作符只有一个, 即?:, 三元操作符有三个操作数且使用中缀符号 (c? x: y)。

在一个表达式中操作符的求值顺序是由这些操作符的 precedence 和 associativity 确定的。

某些操作符可被 overloaded。操作符重载允许在一或两个操作数都是用户自定义类或结构的操作中指定用户自定义操作符工具。

7.2.1 操作符优先与结合性 (Operator Precedence And Associativity)

当一个表达式含有多个操作符时, 操作符 precedence 决定每个操作符的求值顺序。例如, 表达式 X+Y*Z 这样求值 X+(Y*Z), 因为操作符*比+具有更高的优先级。操作符的优先顺序由其相关的语法定义确定。例如, 一个 additive-expression 包括一系列被操作符+或-隔开的 multiplicative-expressions。操作符+和-比操作符*、/以及%的优先级低。

表 7-1 概括了从优先顺序最高到最低的所有操作符。

表 7-1 操作符的优先顺序

参见节	分 类	操 作 符
7.5	Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
7.6	Unary	+ - ! ~ ++x --x (T)x
7.7	Multiplicative	* / %
7.7	Additive	+ -
7.8	Shift	<< >>
7.9	Relational	< > <= >= is as
7.9	Equality	== !=
7.10	Logical AND	&
7.10	Logical XOR	^
7.10	Logical OR	
7.11	Conditional AND	&&
7.11	Conditional OR	
7.12	Conditional	?:
7.13	Assignment	= *= /= %= += -= <<= >>= &= ^= =

当一个操作数两个操作符具有相同的优先顺序时,这些操作符的 associativity 决定操作的顺序:

- 除赋值操作符外,所有的二元操作符都是 left-associative,也就是说操作是从左到右进行的。例如: $X+Y+Z$ 是这样求值的 $(X+Y)+Z$ 。
- 赋值操作符和条件操作符(?:)都是 right-associative,也就是说操作顺序是从右到左的。例如: $X=Y=Z$ 的求值方法是 $X=(Y=Z)$ 。

优先顺序和结合性可用括弧来控制。例如式子 $X+Y*Z$ 的求值方法是先用 Y 乘以 Z ,其结果再与 X 相加,但是,式子 $(X+Y)*Z$ 的求值方法却是先把 X 和 Y 相加,其结果再与 Z 相乘。

7.2.2 操作符重载 (Operator Overloading)

所有的一元或二进制操作符都具有在任何表达式中都可自动获得的预先定义的工具。除预先定义的工具外,用户自定义工具也可以通过在类和结构中声明 operator 而被引入。用户自定义操作符工具总是优先于预先定义的操作符工具;只有当没有可用的用户自定义操作符工具时,才考虑预先定义的操作符工具。

The overloadable unary operators are:

+ - ! ~ ++ -- true false

The overloadable binary operators are:.

+ - * / % & | ^ << >> == != > < >= <=

只有上面所列举的操作符可被重载。成员访问、方法引用或操作符=、&&、//、?、new、

type of 和 is 决不能被重载。

当一个二进制操作符被重载时，相应的赋值操作符也被默认重载。例如，操作符*的重载也是操作符*=的重载。详情见 7.13 节。注意，赋值操作符本身不被重载。赋值总是将一个值的简单的按位求补复制到一变量中。

CAST 操作如 (T) X 通过提供用户自定义转换而被重载。

元素访问如 A (X) 不可重载。然而，索引支持用户自定义索引。

表达式中使用的是操作符符号，而在声明中，使用的则是函数符号。用户自定义操作符声明要求至少有一个参数为含有此操作符声明的类或结构类型。这样，用户自定义操作符不可能具有与预先定义的操作符相同的签名。

用户自定义操作符声明不能修改操作符的句法、优先顺序或结合性。例如，操作符总是是一个二进制操作符，总具有 7.2.1 节中所指定的优先顺序且总是左结合的。

尽管用户自定义操作符可以执行它所申请的任何运算，但绝不允许产生无法预料的结果。例如，operator== 应该平等地比较这两个操作数并返回一个正确的结果。

7.13 节和 7.5 节中对单个操作符的描述详细说明了这些操作符预先定义的工具以及适用于每个操作符的条件规则。这些描述的含义将见下面几部分。

7.2.3 一元操作符重载分解 (Unary Operator Overload Resolution)

在 opx 或 xop 这样的操作中，op 指的是一个可重载的一元操作符，x 是类型 x 的一个表达式，其过程如下：

- 由 x 提供的操作过程 operator op (x) 的候选用户自定义操作符的集合由中的规则来确定。
- 如果候选用户自定义操作符的集合不是空的，那么它就是此操作过程的候选操作符。否则，预先定义的一元 operator op 工具就是这个操作过程的候选操作符。一给定操作符预先定义功能在对这个操作符的说明中被指定。
- 7.4.2 节中的重载分解规则适用于在候选操作符集合中挑选出与自变量列变 (x) 有关的最佳操作符，且这个操作符就是此重载分解过程的结果。如果重载分解不能选出一个最佳操作符，那么错误出现。

7.2.4 二进制操作符重载分解 (Binary Operator Overload Resolution)

在这样一个操作 xopy 中，op 是一个可重载的二进制操作符，x 是类型 x 的一个表达式，y 是类型 y 的一个表达式，则其操作过程如下：

- 确定由 x 和 y 所提供的操作过程 operator op (x, y) 的候选用户自定义操作符集合，这个集合包括分别由 x 和 y 提供的候选操作符，它们的确定用 7.2.5 节中的规则。如果 x 和 y 的类型相同或都是从一普通的基本类型中派生而来的，那么，共享的候选操作符只出现于这两个集合的交叉部分。
- 如果候选用户自定义操作符不是空的，那么它就是这个操作过程的候选操作符集合。否则，预先定义的二进制 operator op 工具就是这个操作的候选操作符的集合。一给定操作符的预先定义的功能在对其描述中被指定。

- 7.4.2 节中的重载分解规则适用于从候选操作符集合中挑选出与自变量列表 (x, y) 有关的最佳操作符, 且这个操作符就是此重载分解过程的结果。如果重载分解不能选出一个最佳操作符, 那么就是错误的。

7.2.5 候选用户自定义操作符 (Candidate User-Defined Operators)

给定一类型 T 和一操作 operator op (A), 这里, op 是一个可重载的操作符, A 是一个自变量列表, 则由 T 提供的操作 operator op (A) 的候选用户自定义操作符集合的确定如下:

- 对于所有的 operator op 声明来说, 如果至少有一个与自变量列表 A 有关的可用的操作符, 那么, 候选操作符的集合包括 T 中所有可用的 operator op 声明。
- 否则, 如果 T 是 object, 那么, 候选操作符集合就是空的。
- 否则, 由 T 提供的候选操作符的集合就是由 T 的直属基类提供的候选操作符的集合。

7.2.6 数提升 (Numeric Promotions)

数提升自动执行某些预先定义的一元和二进制数值操作符的操作数的隐式转换。数提升不是一个独立的操作, 而是在预先定义的操作符中运用重载分解的一个结果。尽管用户自定义操作符在执行过程中也会出现相似的结果, 但数提升并不影响用户自定义操作符的求值。

作为数提升的一个例子, 考虑到二进制操作符*的预先定义的功能:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

这里, 在操作符的集合中应用重载分解规则, 其效果就是要选择来自于此操作数类型的隐式转换的第一个操作符。例如, 对于操作 b*s 来说, 这里, b 是一个 byte, s 是一个 short, 重载分解选择 operator* (int, int) 作为最佳操作符。这样, 其结果就是, b 和 s 被转化为 int, 且结果类型也是 int。同样, 对于操作 i*d 来说, 这里, i 是一个 int, d 是一个 double, 重载分解选择 operator* (double, double) 作为最佳操作符。

7.2.6.1 一元数提升 (Unary Numeric Promotions)

一元数提升的作用对象是预先定义的一元操作符+、-和~的操作数。一元数提升的作用只是把类型 sbyte、byte、short、ushort 或 char 的操作数转化为类型 int。另外, 对于一元操作符来说, 一元数提升把类型 uint 的操作数转化为类型 long。

7.2.6.2 二进制数提升 (Binary Numeric Promotions)

二进制数提升的作用对象是预先定义的二进制操作符+、-、*、/、%、&、/、^、==、!=、>、<、>=和<=的操作数。二进制数提升把两个操作数隐式转化为一普通的类型, 这个类型即

使在无关操作符存在的情况下，也是此操作的结果类型。二进制数提升必须遵循下列规则，按顺序它们是：

- 如果两个操作数中有一个是 decimal 类型，则另一个也要被转化为 decimal 类型，否则，如果另一个操作数是 float 或 double 型，将会出现错误。
- 否则，如果两个操作数中有一个是 double 型，则另一个也要被转化为 double 型。
- 否则，如果两个操作数中有一个是 float 型，则另一个也要被转化为 float 型。
- 否则，如果有一个是 ulong 型，则另一个也要被转化为 ulong 型，否则，如果另一个是 sbyte、short、int 或 long 型，则会出错。
- 否则，如果有一个是 long 型，则另一个也要被转化为 long 型。
- 否则，如果有一个是 uint 型，另一个是 sbyte、short 或 int 型，则两个操作数都要被转化为 long 型。
- 否则，如果有一个是 uint 型，另一个也要被转化为 uint 型。
- 否则，两个操作数都要被转化为 int 型。

注意，在第一条规则中不允许类型 decimal 与 double 和 float 混合操作。这条规则是根据下面的事实制定的，即在 decimal 型和 double 型、float 型之间没有隐式转换。

还要注意，当某一操作数是带符号的整类型时，另一个不能是 ulong 型。原因是，不存在既表示带符号的整类型的取值范围，又表示 ulong 的取值范围的整类型。

在上面两种情况下，都可以用 cast 表达式把一个操作数显式转化为同另一个操作数一致的类型。

下面的例子在编辑时将会出现错误，因为 decimal 不能与 double 相乘：

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

把第二个操作数显式转化为 decimal 型可以消除此错误。

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

7.3 成员查找 (Member Lookup)

成员查找指的是这样一个过程，即通过它可以确定一类型上下文中的某一名字的意思。成员查找可以作为一表达式中 simpli-name 或 member-access 求值的一部分。

类型 T 中的一个名字 B 的成员查找过程如下：

- (1) 首先，构造一个在 T 中声明的已命名为 N 的所有可访问成员以及 T 的基本类型的集合。含有 override 修改函数的声明不在此集合之内。如果不存在名字为 N 的成员或这样的成员不可访问，那么，就找不到要查找的对象，且下面的步骤也不被求值。
- (2) 其次，将被其它成员隐藏的成员清除出这个集合。对这个集合中的每一个成员 S.M 来说，这里 S 位于声明成员 M 的类型中，下面的规则都适用。

- 如果 M 是一个常量、域、属性、事件、类型或枚举成员，那么在 S 的基本类型中声明的所有成员都要被清除这个集合。
- 如果 M 是一个方法，那么，在 S 的基本类型中声明的所有非方法成员都要被清除出这个集合，且在 S 的基本类型中声明的所有与 M 具有相同签名的方法也要被清除出这个集合。

(3) 最后，当清除完被隐藏成员以后，查找的结果确定如下：

- 如果一个集合含有一个非方法成员，那么，这个成员就是查找的结果。
- 否则，如果一个集合只包括方法，那么，这个方法群就是查找的结果。
- 否则，查找就是不清楚的，编辑时将出现错误（这种情况只发生在含有多个直属基本接口的接口成员查找过程中）。

对于非接口类型或严格单继承接口（继承链中的每一个接口都只有 0 或 1 个直属基本接口）中的成员查找来说，这种查找规则的效果是派生的成员用同样的名字或签名隐藏基本成员。

为方便成员查找，认为类型 T 的基本类型（Base Types）分为以下几种情况：

- 如果 T 是 object，那么，T 没有基本类型。
- 如果 T 是一个 value-type，那么 T 的基本类型就是类类型 object。
- 如果 T 是一个 class-type，那么 T 的基本类型就是 T 的基类，包括类类型 object。
- 如果 T 是一个 interface-type，那么 T 的类型就是 T 的基本接口及类类型 object。
- 如果 T 是一个 array-type，那么 T 的基本类型就是类类型 system、array 以及 object。
- 如果 T 是一个 delegates-type，那么 T 的基本类型就是类类型 system、delegate 以及 object。

7.4 函数成员（Function Members）

函数成员就是指那些含有可执行语句的成员。函数成员总是类型成员而不会是名字空间成员。C# 中的函数成员有五类：

- 构造函数。
- 方法。
- 属性。
- 索引。
- 用户自定义操作符。

函数成员所含有的语句通过函数成员引用来执行。函数成员书写的实际句法取决于特定的函数成员。然而，所有的函数成员引用都是表达式，允许将自变量传到函数成员中，且允许函数成员运算并返回一个结果。

函数成员引用中的自变量列表为该函数成员的参数提供实际的值或变量引用。

构造函数、方法、索引和操作符的引用通过重载分解来确定引用函数成员候选集中的哪一个。这个过程将在 7.4.2 节中进行描述。

一特定的函数成员在编辑期被识别（可能通过重载分解）后，函数成员引用的实际运行期过程将在 7.4.3 节中描述。

表 7-2 包括五种函数成员,概括了发生在结构中的过程。在这个表格中,E、X、Y 和 VALUE 指的是作为变量或值的表达式, T 表示一作为类型的表达式, F 是一方法的简化名, P 是一属性的简化名。

表 7-2 函数成员的种类

结 构	例 子	说 明
构造函数引用	new T(x,y)	用重载分解从给定的类或结构 T 中选择最佳构造函数, 这个构造函数在引用时带有自变量列表 (x,y)
方法引用	F(x,y)	用重载分解从上级类或结构中选择最佳方法 F, 此方法在引用时带有自变量列表 (x,y), 如果这个方法不是 static, 那么, 其实例表达式就是 this
	T.f(x,y)	用重载分解从类或结构 T 中选择最佳方法 F, 如果这个方法不是 static, 则错误发生。此方法在引用时带有自变量列表(x,y)
	e.F (x, y)	用重载分解从类型 E 所给定的类、结构或接口中选择最佳方法 F, 如果这个方法是 static, 那么将会出现错误。此方法在引用时具有实例表达式 e 和自变量列表 (x,y)
属性访问	P	引用上级类或结构中属性 P 的访问函数 get, 如果 P 为只写, 则错误发生。如果 P 不是 static, 则其实例表达式为 this
	P=value	上级类或结构中属性 P 的 set 访问函数同其自变量列表 (value) 一起被引用, 如果 P 为只写, 则错误发生。如果 P 不是 static, 则其实例表达式为 this
	T.P	引用类或结构 T 中属性 P 的访问函数 get, 如果 P 不是 static 或 P 为只写, 则错误发生
	T.P=value	引用类或结构 T 中属性 P 的访问函数 set 及其自变量列表 (value), 如果 P 不是 static 或 P 为只读, 则错误发生
	e.P	引用由类型 e 给定的类、结构或接口中属性 P 的访问函数 get 及实例表达式 e, 如果 P 为 static 或 P 为只写, 则错误发生
	e.P=value	由类型 e 给定的类、结构或接口中属性 P 的访问函数 set 在被引用时带有实例表达式 e 及自变量列表 (value)。如果 P 为 static 或 P 为只读, 则错误发生
索引访问	e[x,y]	用重载分解从类型 E 给定的类、结构或接口中选择最佳索引函数, 引用此索引函数的 get 访问函数及实例表达式 e 和自变量列表 (x,y), 如果此索引函数为只读, 则错误发生
	e[x,y]=value	用重载分解从类型 E 所给定的类、结构或接口中选择最佳索引函数, 引用此索引函数的 set 访问函数及实例表达式 e 和自变量列表 (x,y,value), 如果此索引函数为只读, 则错误发生
操作符引用	-x	用重载分解从由类型 x 给定的类或结构中选择最佳一元操作符, 引用所选操作符及其自变量列表 (x)
	x+y	用重载分解从由类型 x 和 y 给定的类或结构中选择最佳二元操作符, 引用所选操作符及其自变量列表 (x, y)

7.4.1 自变量列表 (Argument Lists)

每一个函数成员引用都包括一个自变量列表。自变量列表为函数成员参数提供实际值或变量引用。函数成员引用自变量列表的句法取决于函数成员的种类:

- 对于构造函数、方法和委托来说, 正如下面所述, 自变量被作为 argument-list 来说明。
- 对于属性来说, 当引用访问函数 get 时, 自变量列表是空的, 当引用访问函数 set

时，自变量列表指的是作为赋值操作符右操作数的表达式。

- 对于索引来说，自变量列表指的是索引访问中两个方括号之间的表达式。当引用访问函数 set 时，自变量列表包括作为赋值操作符右操作数的表达式。
- 对于用户自定义操作符来说，自变量列表包括一元操作符的单个操作数或二进制操作符的两个操作数。

属性、索引以及用户自定义操作符的自变量总是被作为值参数而传递。这些函数成员不支持引用参数和输出参数。

构造函数、方法或委托引用的自变量被列为同一个 argument-list:

```
argument-list:
    argument
    argument-list , argument

argument:
    expression
    ref variable-reference
    out variable-reference
```

一个 argument-list 包括 0 或多个被逗号隔开的 arguments。自变量的形式有以下几种:

- 表达式，指的是作为值参数传递的自变量。
- 带有一个 variable-reference 的关键词 ref，指的是作为引用参数而传递的自变量。在函数成员引用中，如果一个变量作为输出参数传递，那么，在其函数成员被引用之后就认为这个变量已被明确赋值。

在函数成员引用过程中，自变量列表的表达式或成员引用从左到右按顺序求值，具体情况如下:

- 对于一个值参数来说，对自变量表达式进行求值并执行一个到相应参数类型的隐式转换。其结果值就是函数成员引用中这个值参数的初始值。
- 对于一个引用或输出参数来说，对变量引用进行求值，所得的存储空间就是函数成员引用中这个参数所表示的存储空间。如果给定作为引用或输出参数的变量引用是 reference-type 的数组元素，那么，在运行期一定要进行检查以确保这个数组的元素类型同此参数的类型一致。如果检查失败，则显示 ArrayTypeMismatch Exception。

方法、索引以及构造函数可以将其上一个参数声明为参数数组。这样的函数成员以它们正常的形式被引用还是以其扩展形式被引用取决于哪一个可行。

- 当带有一参数数组的函数成员以其正常形式被引用时，这个参数数组所给定的自变量必须是一类型的单个表达式，此类型可被隐式转化为上述参数数组的类型。在这种情况下，这样的参数数组就相当于一个值参数。
- 当带有一参数数组的函数成员以其扩展形式被引用时，此引用必须为这个参数数组指定 0 或多个自变量，这里，每一个自变量都是一个表达式，其类型可被隐式转化为参数数组的元素类型。在这种情况下，此引用产生上述参数数组类型的一个实例，其长度与自变量的数目相等，此引用还用所给的自变量的值对上述数组实例的元素进行初始化并把新产生的数组实例作为实际自变量来应用。

一个自变量列表的表达式总是以书写顺序被求值。因此，下面的例子输出的结果为：x=0，y=1，z=2

```
class test
{
    static void f(int x, int y, int z) {
        console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
    }
}
```

数组方差规则允许数组类型 A[] 的值被数组类型 B[] 的一个实例所引用，假设从 B 到 A 存在一个隐式引用转换。由于这些规则，当 reference-type 的数组元素被作为引用或输出参数传递时，需要进行运行期检查以确保这个数组的实际元素类型同此参数的实际元素类型一致。在下面的例子中：

```
class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);           // Ok
        F(ref b[1]);           // ArrayTypeMismatchException
    }
}
```

第二个 F 的引用导致显示 ArrayTypeMismatchException，因为 B 的实际元素类型是 string 而不是 object。

当带有一个参数数组的函数成员以其扩展形式被引用时，此引用过程就相当于把一带有数组初始化函数的数组产生的表达式插入到这些扩展参数的周围。例如，给定一个声明：

```
void f (intx, inty, params object[] args);
```

下面这个方法扩展形式的引用

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

与下面的内容一致：

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

注意：特殊情况下，当给这个参数数组以零自变量时，产生一个空的数组。

7.4.2 重载分解 (Overload Resolution)

重载分解指的是选择最佳函数成员以引用给定自变量列表以及候选函数成员集合的一种方法。C#中，重载分解所选择的函数成员应用于下列特定的上下文中：

- 在 invocation-expression 中命名的一个方法的引用过程中。
- 在 object-creation-expression 中命名的一个构造函数的引用过程中。
- 通过 element-access 的一个索引访问函数的引用过程。
- 在一表达式中引用的一预先定义或用户自定义操作符的引用过程。

这些上下文每一个都以其特定的方式定义了候选函数成员的集合以及自变量列表。然而，一旦候选函数成员及自变量列表被确定，在任何情况下，最佳函数成员的选择过程都是一样的：

- 首先，筛选候选函数成员的集合，只保留那些与给定自变量列表有关的可用函数成员。如果这个筛选后的集合是空的，则错误出现。
- 其次，给定可用候选函数成员的集合，并从中找出最佳函数成员。如果这个集合只有一个函数成员，那么，这个函数成员就是最佳函数成员。否则，最佳函数成员就是与给定自变量列表有关的比其它所有函数成员都好的那个函数成员，假设依据 7.4.2.2 节中的规则，每一个函数成员与所有其它函数成员都具有可比性。如果不存在一个比其它所有函数成员都好的函数成员，那么，此函数成员引用就是模糊的，错误的。

下面几部分解释了术语 applicable function member 以及 better function member 的确切含义。

7.4.2.1 可用函数成员 (Applicable Function Member)

当一个函数满足下列所有条件时，这个函数就是一个与自变量列表 A 有关的 applicable function member：

- A 中的自变量数与此函数成员声明中的参数数量一致。
- 对于 A 中的每一个自变量来说，传递其模式的参数与传递相应参数模式的参数一致，且对于值参数或参数数组来说，存在从其自变量类型到相应参数类型的隐式转换。或者对于参数 ref 或 out 来说，其自变量类型与相应参数的类型一致。

对于一个含有参数数组的函数成员来说，如果依据上面的规则，这个函数成员是可用的，就说它以其正常形式是可用的。如果一含有参数数组的函数成员以其正常形式是不可用的，那么这个函数成员可能以其扩展形式是可用的：

- 扩展形式是这样形成的，即用该参数数组元素类型的 0 或多个值参数代替函数成员声明中的参数数组，以使自变量列表 A 中的自变量数与整个参数数相等。如果 A 的

自变量数比函数成员声明中的固定参数数少，那么这个函数成员的扩展形式就不能形成。因此，也就不可用。

- 如果声明函数成员的类、结构或接口已含有另一个与此扩展形式具有相同签名的函数成员，那么，这个扩展形式也是不可用的。
- 否则，如果对 A 中的每一个自变量来说，传递其模式的参数与传递相应参数模式的参数一致，那么，这个扩展形式就是可用的，且：
 - 对于一固定的或由扩展而产生的值参数来说，存在从其自变量类型到相应参数类型的隐式转换，或者
 - 对于参数 ref 或 out 来说，自变量的类型与相应参数的类型一致。

7.4.2.2 较好函数成员 (Better Function Member)

给定一具有自变量类型集合 A1, A2……, An 的自变量列表 A 以及两个分别具有参数类型 P1, P2, ……Pn 和 Q1, Q2……Qn 的函数成员 Mp 和 Mq，如果下列条件符合，那么，就说 Mp 是一个比 Mq 较好的函数成员：

对于每一个自变量来说，从 Ax 到 Px 的隐式转换并不比从 Ax 到 Qx 的隐式转换差，且对于至少一个自变量来说，从 Ax 到 Px 的转变比从 Ax 到 Qx 的转换好。

在执行这样的求值时，如果 Mp 或 Mq 以其扩展形式是可用的，那么，Px 或 Qx 指的就是这个参数列表的扩展形式的一个参数。

7.4.2.3 较好转换 (Better Conversion)

给定一个从类型 S 到类型 T1 的隐式转换 C1 以及从类型 S 到类型 T2 的隐式转换 C2，那么，这两个转换的 **better conversion** 确定如下：

- 如果 T1 和 T2 的类型相同，则两个转换效果相同。
- 如果 S 是 T1，那么 C1 就是较好转换。
- 如果 S 是 T2，那么 C2 就是较好转换。
- 如果存在从 T1 到 T2 的隐式转换，而不存在从 T2 到 T1 的隐式转换，那么 C1 就是较好转换。
- 如果存在从 T2 到 T1 的隐式转换，而不存在从 T1 到 T2 的隐式转换，那么 C2 就是较好转换。
- 如果 T1 是 sbyte 而 T2 是 byte、ushort、uint 或 ulong，那么，C1 就是较好转换。
- 如果 T2 是 sbyte 而 T1 是 byte、ushort、uint 或 ulong，那么，C2 就是较好转换。
- 如果 T1 是 short 而 T2 是 ushort、uint 或 ulong，那么，C1 就是较好转换。
- 如果 T2 是 short 而 T1 是 ushort、uint 或 ulong，那么，C2 就是较好转换。
- 如果 T1 是 int 而 T2 是 uint 或 ulong，那么，C1 就是较好转换。
- 如果 T2 是 int 而 T1 是 uint 或 ulong，那么，C2 就是较好转换。
- 如果 T1 是 long 而 T2 是 ulong，那么，C1 就是较好转换。
- 如果 T2 是 long 而 T1 是 ulong，那么，C2 就是较好转换。
- 否则，两个转换效果相同。

如果依据这些规则，隐式转换 C1 是比隐式转换 C2 较好的转换，那么，C2 就是一个比 C1 较差的转换。

7.4.3 函数成员引用 (Function Member Invocation)

这部分描述的是运行期引用一特定函数成员的过程。假设编辑期已确定引用这个特定成员（可能通过在候选函数成员集合中运用重载确定）。

为方便说明此引用过程，把函数成员分为以下两类：

- 静态函数成员，它们是静态方法、构造函数、静态属性访问函数以及用户自定义操作符。静态函数成员都是非虚拟的。
- 实例函数成员，它们是实例方法、实例属性访问函数以及索引访问函数。实例函数成员有的是虚拟的，有的是非虚拟的，但都在一特定的实例中被引用。实例通过实例表达式参与运算，且它在 `this` 这样的函数成员中是可访问的。

函数成员引用的运行期过程包括以下几步，其中，`M` 是这个函数成员，且如果 `M` 是一个实例成员，那么 `E` 就是相应的实例表达式：

- 如果 `M` 是一个静态函数成员：
 - 引用 `M`。
 - 如果 `M` 是在 `value-type` 中声明的一个实例函数成员：
- 对 `E` 求值，如果求值过程出现异常，则执行停止。
 - 如果 `E` 不是一个变量，则产生一个类型为 `E` 的临时局部变量，且 `E` 的值被赋给那个变量。这时，`E` 就成了这个临时局部变量的一个引用。此临时变量作为 `M` 中的 `this` 是可访问的，但在其它情况下是不可访问的。因此，只有当 `E` 是一个真正的变量时，调用者才可观察到 `M` 所导致的 `this` 的变化。
 - 对自变量列表进行求值，方法同 7.4.1 节中所述。
 - 引用 `M`，则被 `E` 引用的变量就成了被 `this` 所引用的变量。
 - 如果 `M` 是一个在 `reference-type` 中声明的实例函数成员，那么：
- 对 `E` 求值，如果求值过程出现异常，则执行停止。
 - 对自变量列表进行求值，方法同 7.4.1 节中所述。
 - 如果 `E` 的类型是 `value-type`，则执行一个从 `E` 到类型 `object` 的封箱转换，且在下面的步骤中，`E` 的类型被认为是 `object`。
 - `E` 的值经检查是有效的。如果 `E` 的值是 `null`，则显示 `NullReferenceException`，执行停止。
 - 确定要引用的函数成员工具。如果 `M` 是一个非虚拟的函数成员，那么，`M` 就是要引用的函数成员工具。否则，如果 `M` 是一个虚拟的函数成员，那么，就要通过虚拟函数成员查找或接口函数成员查找来确定要引用的函数成员工具。
 - 引用在以上步骤中所确定的函数成员工具。那么，被 `E` 所引用的对象就成了被 `this` 所引用的对象。

下列情况下，在 `value-type` 中执行的函数成员可以通过该 `value-type` 的封装实例而被引用：

- 当此函数成员是从类型 `object` 继承的一个方法的 `override`，且通过其类型 `object` 的实例表达式而被引用时。
- 当此函数成员是一接口函数成员的工具，且通过 `interface-type` 的一个实例表达式而被引用时。

- 当此函数成员通过一个委托而被引用时。

在这些情况下，就认为封装实例包括 `value-type` 的一个变量，且这个变量就是在函数成员引用中被 `this` 所引用的变量。这就意味着，在一定情况下，当一个函数成员在一封装实例中被引用时，它可能修改这个封装实例所包含的值。

7.5 原始表达式 (Primary Expressions)

原始表达式(primary-expression)包括以下种类：

```
literal
simple-name
parenthesized-expression
member-access
invocation-expression
element-access
this-access
base-access
post-increment-expression
post-decrement-expression
new-expression
typeof-expression
sizeof-expression
checked-expression
unchecked-expression
```

7.5.1 字母 (Literals)

含有一个 `literal` 的 `primary-expression` 就是一个值。

7.5.2 简化名 (Simple Names)

一个 `simple-name` 包括一单个识别符号。

```
simple-name:
identifier
```

`simple-name` 的求值和分类情况如下：

- 如果一个 `simple-name` 出现在一个 `block` 中且这个 `block` 包括一给定名字的局部变量或参数，那么，这个 `simple-name` 指的就是那个局部变量或参数且被归类为一个变量。
- 否则，对于每一个类型 `T` 来说，都开始于它所直属的类、结构或枚举的声明，紧接着是每一个上级外部类或结构的声明（如果有的话），如果 `T` 中 `simple-name` 的成员查找产生一个对象：

- 如果 T 就是其所直属的类或结构类型，且此查找能识别一个或多个方法，那么，其结果就是一带有 this 的相关实例表达式的方法群。
- 如果 T 就是其所直属的类或结构类型，且此查找能识别一个实例成员，且引用发生在一构造函数、实例方法或实例访问函数的块内，那么，其结果就是式子 THIS.E 的一个成员访问，这里 E 就是这个 simple-name。
- 否则，其结果就与式子 T · E 的成员访问相同，这里，E 是 simple-name，在这种情况下，simple-name 引用实例成员是错误的。
- 否则，对于每一个类型 T 来说，开始产生 simple-name 的名字空间声明，然后是每一个上级名字空间声明（如果有），最后结束于总名字空间，对下列步骤进行求值，直至找到一个实体：
 - 如果一个名字空间包含一给定名字的名字空间成员，那么，其 simple-name 指的就是那个成员且依据这个成员而被归类为名字空间或类型。
 - 否则，如果一个名字空间包括连接给定名字及引进名字空间或类型的 using-alias-directive，那么，其 simple-name 指的就是那个名字空间或类型。
 - 否则，如果由名字空间声明的 using-name space-directive 输入的名字空间只含有一个给定名字的类型，那么，其 simple-name 就是指的那个类型。
 - 否则，如果由名字空间声明的 using-name space-directive 输入的名字空间含有多一个给定名字的类型，那么，其 simple-name 就是模糊的，错误出现。
- 否则，由 simple-name 给定的名字就是未定义的，错误出现。

对于表达式中每一个给定作为 simple-name 的标识符来说，此标识符在相邻的上级 block 或 switch-block 内的表达式中作为 simple-name 的每一次出现都必须指的是相同的实体。这条规则确保表达式上下文中某一个名字的意思在同一块内总是相同的。

下面的例子就是错误的，因为 X 在外部范围的块内指的是不同的意思（外部块包括 IF 语句中的嵌套块）。

```
class Test
{
    double x;

    void F(bool b) {
        x = 1.0;
        if (b) {
            int x = 1;
        }
    }
}
```

而下面的例子则是正确的，因为名字 X 从未在外部块中使用。

```
class Test
{
```

```
double x;
void F(bool b) {
    if (b) {
        x = 1.0;
    }
    else {
        int x = 1;
    }
}
}
```

注意：不变意义规则只适用于简化名。对于同一个标识符来说，下列情况是有效的，即它作为简化名时是一种意思，而作为成员访问的右操作数时又是另一种意思。

例如：

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

上面的例子是把域名用作构造函数参数名的一个常见模式。在这个例子中，简化名 `x` 和 `y` 指的是参数，但它并不妨碍成员访问表达式 `THIS.X` 和 `THIS.Y` 访问这个域。

7.5.3 括弧表达式 (Parenthesized Expressions)

`parenthesized-expression` 指的是括弧内的 `expression`。

```
parenthesized-expression:
    ( expression )
```

对 `parenthesized-expression` 进行求值也就是对括弧内的表达式进行求值，如果括弧内的 `expression` 指的是名字空间、类型或方法群，则错误出现。否则，`parenthesized-expression` 的结果就是对其所含的 `express` 求值的结果。

7.5.4 成员访问 (Member Access)

`member-access` 包括一个 `primary-expression` 或 `predefined-type`，其后是符号“`·`”，然后是一个 `identifier`。

```

member-access:
    primary-expression . identifier
    predefined-type . identifier

predefined-type: one of
    bool byte char decimal double float int long
    object sbyte short string uint ulong ushort

```

在式子 E.I 的 member-access 中, E 是 primary-expression 或 predefined-type, I 指的是一个 identifier, 其求值和分类情况如下:

- 如果 E 是一个名字空间, I 是那个名字空间的一个可访问成员的名字, 那么, 其结果就是那个成员, 并依据那个成员而被归类为一个名字空间或一个类型。
- 如果 E 是一个作为类型的 predefined-type 或 primary-expression, 且 E 中 I 的一个成员查找产生一个匹配值, 那么 E, I 的求值及分类如下:
 - 如果 I 识别一个类型, 那么, 其结果就是那个类型。
 - 如果 I 识别一个或多个方法, 那么, 其结果就是与实例表达式无关的一个方法群。
 - 如果 I 识别一个 static 属性, 则其结果就是与实例表达式无关的一个属性。
 - 如果 I 识别一个 static 域:

如果这个域是 readonly, 且引用发生在声明这个域类或结构的静态构造函数之外, 那么, 其结果就是 E 中静态域 I 的一个值。否则, 其结果就是 E 中的静态域 I 的一个变量。
 - 如果 I 识别一个 static 事件:

如果引用发生在声明这个事件的类或结构之内, 那么, 在对 E.I 进行操作时, 就把 I 当作一个静态域或属性来使用。否则, 其结果就是一个与实例表达式无关的事件访问。
 - 如果 I 识别一个常量, 那么, 其结果就是该常量的一个值。
 - 如果 I 识别一个枚举成员, 则其结果就是该枚举成员的一个值。
 - 否则, E.I 就是一个无效的成员引用, 错误出现。
- 如果 E 是一类型为 T 的属性访问、索引访问、变量或值, 且 T 中 I 的成员查找产生一个匹配值, 那么 E.I 的求值及分类情况如下:
 - 首先, 如果 E 是一个属性或索引访问, 那么就可得到这个属性或索引访问的值, 且 E 被重新归类为一个值。
 - 如果 I 识别一个或多个方法, 则结果就是一个方法群, 这个方法群带有 E 的一个相关实例表达式。
 - 如果 I 识别一个实例属性, 则其结果就是带有一个 E 中相关实例表达式的属性访问。
 - 如果 I 是一个 class-type, 且 I 识别该 class-type 的一个实例域:

如果 E 的值是 null, 则显示 NullReference Exception。否则, 如果这个域是 readonly, 且引用发生在声明这个域的类的一个实例构造函数之外, 则其结果就是被 E 引用的对象中域 I 的一个值。否则, 结果就是被 E 引用的对象中域 I

的一个变量。

- 如果 I 是一个 struct-type, 且 I 识别该 struct-type 的一个实例域:
如果 E 是一个值或此域为 readonly, 且引用发生在声明这个域的结构的一个实例构造函数之外, 则结果就是由 E 给定的结构实例中域 I 的一个值。
否则, 结果就是一个变量, 这个变量就是由 E 给定的结构实例的域 I。
 - 如果 I 识别一个实例事件, 则:
如果引用发生在声明这个事件的类或结构内, 则在对 E.I 进行操作时, 就把 I 当作一个实例域或属性来引用。否则, 结果就是一个事件访问, 这个访问带有 E 中一个相关的实例表达式。
- 否则, E.I 就是一个无效的成员引用, 错误发生。

在式子 E.I 的成员访问中, 如果 E 是一个标识符, 且 E 作为 simple-name (参见 7.5.2 节) 的意思是一个常量、域、属性、局部变量或一个与作为 type-name (参见 3.8 节) 的 E 意思类型相同的参数, 那么, E 就有两种可能的意思。E.I 的这两种意思不可能混淆, 因为在两种情况下, I 都必须是类型 E 的成员。也就是说, 这条规则只允许访问 E 的静态成员, 否则, 错误出现。例如:

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);
    public Color Complement() {...}
}

class A
{
    public Color Color;                // Field Color of type Color
    void F() {
        Color = Color.Black;          // References Color.Black static member
        Color = Color.Complement();   // Invokes Complement() on Color field
    }
    static void G() {
        Color c = Color.White;        // References Color.White static member
    }
}
```

类 A 中, 在标识符 Color 出现的地方, 凡是引用类型是 Color 的, 都被标上了下划线。

7.5.5 引用表达式 (Invocation Expressions)

引用表达式 (invocation-expression) 的作用是引用一个方法。

invocation-expression:

`primary-expression (argument-listopt)`

`invocation-expression` 的 `primary-expression` 必须是方法群或 `delegate-type` 的一个值。如果 `primary-expression` 是一个方法群, 则 `invocation-expression` 就是一个方法引用 (参见 7.5.5.1 节)。如果 `primary-expression` 是 `delegate-type` 的一个值, 则 `invocation-expression` 就是一个委托引用 (参见 7.5.5.2 节)。如果 `primary-expression` 既不是一个方法群也不是 `delegate-type` 的一个值, 则错误出现。

可选择的 `argument-list` (参见 7.4.1 节) 为方法参数提供值或变量引用。对 `invocation-expression` 求值的结果可分为以下几类:

- 如果这个 `invocation-expression` 引用一返回值为 `void` 的方法或委托, 则结果为空。空表达式不能作为任何操作符的操作数, 且只允许出现在 `statement-expression` (参见 8.6 节) 的上下文中。
- 否则, 结果就是由这个方法或委托返回的一个值。

7.5.5.1 方法引用 (Method Invocations)

对于一个方法引用来说, `invocation-expression` 的 `primary-expression` 必须是一个方法群。这个方法群识别一个要引用的方法或被重载方法的集合, 从这个集合中可以选择一特定的方法来引用。在后一种情况下, 要引用的特定方法根据 `argument-list` 中自变量的类型所提供的上下文来确定。

在式子 `M(A)` 中, `M` 指的是一个方法群, `A` 表示一个可选择的 `argument-list`, 它的方法引用的编辑期处理包括以下几步:

- 构建这个方法引用的候选方法集合。这个集合开始于与 `M` 有关的方法的集合, 此集合可以通过前面的成员查找 (参见 7.3 节) 而找到, 然后对这个集合进行筛选, 只保留那些与自变量有关的可用方法。此集合的筛选包括对集合中每一个方法 `T.N` 依据下面的规则进行处理, 这里, `T` 指的是声明方法 `N` 的那个类型:
 - 如果 `N` 对 `A` 无用 (参见 7.4.2.1 节), 则 `N` 将被清除出这个集合。
 - 如果 `N` 对 `A` 有用, 则在 `T` 的基本类型中声明的所有方法都将被清除出这个集合。
- 如果所得到的候选方法集合是空的, 则不存在可用的方法, 错误出现。如果这些候选方法不是在同一个类型中声明的, 则这个方法引用就是模糊的错误出现 (后一种情况只发生在接口中的方法引用过程中, 此接口具有多个直属的基本接口, 详情参见 13.2.5 节)。
- 候选方法集合中的最佳方法是通过 7.4.2 节中的重载分解规则而确定的。如果不能确定一单个最佳方法, 则这个方法引用就是不清楚的, 错误出现。
- 给定一个最佳方法, 则这个方法的引用在此方法群上下文中是合法的。如果这个最佳方法是一个静态方法, 则这个方法群一定是按照类型从 `simple-name` 或 `member-access` 中得到的。如果这个最好方法是一个实例方法, 则这个方法群一定是按照变量或值或 `base-access` 从 `simple-name` 或 `member-access` 中得到的, 如果这两个要求都不符合, 则错误出现。一旦一个方法通过上面的步骤被选定且在编辑期内合法, 则实际运行期引用过程将依据 7.4.3 节中的函数成员引用规则来进行。

上面所述分解规则的直观效果如下: 找到被一方法引用过程所引用的特定方法, 从这个

方法引用所示的类型开始，然后是一个继承链直至至少找到一个可用且访问的有效的方法声明。然后对在那个类型中声明的可用且可访问的有效方法进行重载分解，并引用所选择的方法。

7.5.5.2 委托引用 (Delegate Invocations)

对于一个委托引用来说，invocation-expression 的 primary-expression 必须是 delegate-type 的值。而且，由于此 delegate-type 是一个与 delegate-type 具有相同参数列表的函数成员，因此，这个 delegate-type 对 invocation-expression 的 argument-list 必须是有用的（参见 7.4.2.1 节）。

在式子 D (A) 中，D 表示一个 delegate-type 的 primary-expression，A 表示一个可选的 argument-list，其委托引用的运行期处理过程包括以下几步：

- 对 D 求值，如果求值过程出现异常，则执行停止。
- 检查 D 值的有效性，如果 D 的值是 null，则显示 NullReferenceException，且执行停止。
- 如果 D 是一个委托实例的引用，则在这个委托所引用的方法中执行一个函数成员引用（参见 7.4.3 节）。如果此方法是一个实例方法，则这个引用的实例就被此委托引用。

7.5.6 成员访问 (Element Access)

成员访问 (element-access) 包括一个 primary-expression，其后是符号 “[”，然后是一个 expression-list，最后是符号 “]”。这里，expression-list 包括一个或多个由逗号隔开的表达式。

```

element-access:
    primary-expression [ expression-list ]

expression-list:
    expression
    expression-list , expression
  
```

如果一个 element-access 的 primary-expression 是 array-type 的一个值，则这个 element-access 就是一个数组访问（参见 7.5.6.1 节）。否则，这个 primary-expression 必须是一个变量或值，这个变量或值具有一个或多个索引成员的类、结构或接口类型，且 element-access 就是一个索引访问（参见 7.5.6.2 节）。

7.5.6.1 数组访问 (Array Access)

对于一个数组访问来说，element-access 的 primary-expression 必须是 array-type 的一个值。Expression-list 中的表达式数必须与 array-type 的等级一致，且每一个表达式的类型必须是 int、uint、long、ulong 或可被隐式转化为一个或多个这些类型的类型。

对一个数组访问进行求值的结果是这个数组元素类型的一个变量，称为被 expression-list 中表达式的值所选中的数组元素。

在式子 P (A) 中，P 表示 array-type 的一个 primary-expression，A 是一个 expression-list，其数组访问的运行期处理步骤如下：

- 对 P 求值，如果求值出现异常，则执行停止。

- 对这个 **expression-list** 的索引表达式从左到右按顺序求值。对每一个索引表达式求值之后，都将执行一个隐式转换（参见 6.1 节）转化为下列类型中的一种：**int**、**uint**、**long**、**ulong**。选择存在隐式转换的列表中的第一种类型。例如，如果这个索引表达式是 **short** 型，则将执行一个到 **int** 型的隐式转换，因为从 **short** 到 **int** 以及从 **short** 到 **long** 的隐式转换是可能的。如果一个索引表达式的求值或其后的隐式转换出现异常，则不再对其后的索引表达式进行求值，执行停止。
- 检查 **P** 值的有效性，如果 **P** 的值是 **null**，则显示 **NullReferenceException**，且其它步骤也不再执行。
- 检查 **expression-list** 中每一个表达式的值，以防止 **P** 所引用的数组实例范围的变化。如果有一个或多个值在取值范围之外，则显示 **Index Out of Range Exception**，其它步骤也示再执行。
- 计算由索引表达式给定的数组元素的地址，这个地址就是数组访问的结果。

7.5.6.2 索引访问 (Indexer Access)

对于索引访问来说，其 **element-access** 的 **primary-expression** 必须是类类型、结构类型或接口类型的一个变量或值，且这个类型必须执行一个或多个与这个 **element-access** 的 **expression-list** 有关的可用索引。

在式子 **p(A)** 中，**P** 指的是类、结构或接口类型 **T** 的一个 **Primary-expression**，**A** 是一个 **expression-list** 其索引访问的编辑期处理过程包括以下几步：

- 构建由 **T** 所提供的索引的集合。这个集合包括在 **T** 或 **T** 的一个基本类型中声明的所有索引，这个基本类型不是 **override** 声明且在当前上下文（3.5 节）中可访问。
- 精减这个集合，只保留那些可用的且不被其它索引隐藏的索引。下面的规则适用于这个集合中的每一个索引 **S.I**，这里 **S** 是索引 **I** 声明的类型：
 - 如果 **I** 对 **A**（参见 7.4.2.1 节）来说是不可用的，那么，它将被清除出这个集合。
 - 如果 **I** 对 **A**（参见 7.4.2.1 节）来说是可用的，则在 **S** 的一个基本类型中声明的所有索引都被清除出这个集合。
- 如果所得候选索引集合是空的，则不存在可用的索引，错误出现。如果候选索引不是在同一类型中声明的，则这个索引访问就是不清楚的，将出现错误（后一种情况只发生在对一个具有多个直属基本接口的接口的一个实例的索引访问过程中）。
- 这个候选索引集合的最佳索引是应用 7.4.2 节中的重载分解规则而确定的。如果不能确定一个最佳索引，那么，这个索引访问就是模糊的，错误出现。
- 索引访问处理的结果是一个被归类为访问的表达式。这个索引访问表达式引用在上步中所确定的索引，且具有 **P** 的相关实例表达式以及 **A** 的相关自变量列表。
- 依据应用索引访问的上下文，这个索引访问引用该索引的 **get-accessor** 或 **set-accessor**。如果这个索引访问是一个赋值对象，则就用 **set-accessor** 来赋一个新值（参见 7.13.1 节）。在所有其它情况下，引用 **get-accessor** 得到当前值（参见 7.1.1 节）。

7.5.6.3 字符串索引 (String Indexing)

类 **string** 执行一个索引，这个索引允许一个字符串的单个字符被访问。类 **string** 的索引

具有以下声明：

```
Public char this [int index]{get;}
```

也就是说，一个只读索引采用类型 `int` 的一个自变量，而返回类型 `char` 的一个元素。传递给这个 `index` 自变量的值必须大于或等于零而小于这个字符串的长度。

7.5.7 This 访问 (This Access)

一个 `this-access` 包括专用词 `this`。

`This-access:`

`This`

`this-access` 只允许出现在构造函数实例方法或实例访问的块中。它有以下几种意思：

- 当 `this` 被用在一个类的构造函数的 `primary-expression` 中时，它表示的是一个值。这个值的类型就是发生此引用的类，且这个值是对被构造对象的一个引用。
- 当 `this` 被用在一个类的实例方法或实例访问函数中的 `primary-expression` 中时，它表示一个值。这个值的类型就是发生此引用的类，且这个值就是对引用该方法或访问函数对象的一个引用。
- 当 `this` 被用在一个结构的构造函数内的 `primary-expression` 中时，它表示一个变量。这个变量的类型就是发生此引用的结构，且这个变量表示的是正在被构建的那个结构。一个结构的构造函数的变量就相当于这个结构类型的一个 `out` 参数——意思是说，在这个构造函数的每条执行路径中，此变量都必须被明确赋值。
- 当 `this` 被用在一个结构的实例方法或实例访问函数的 `primary-expression` 中时，它表示一个变量，这个变量的类型就是一个结构，引用就是在这个结构内发生的，且这个变量表示的就是引用这个方法或访问函数的结构。一个结构的实例方法的 `this` 变量就相当于这个结构类型的 `ref` 参数。

`this` 在上面所列的上下文之外的 `primary-expression` 中的运用都是错误的，尤其是不能在一个域声明的静态方法、静态属性访问函数或 `variable-initializer` 中被引用。

7.5.8 基本访问 (Base Access)

一个 `base-access` 包括专用词 `base`，其后是符号 “.” 及一个标识符或包括在方括号内的一个 `expression-list`：

```
base-access:
    base . identifier
    base [ expression-list ]
```

`base-access` 的作用是访问基类成员，这些基类成员被当前类或结构中具有相似名字的成员隐藏。`Base-access` 只允许出现在构造函数、实例方法或实例访问函数的 `block` 中。

当 `base.I` 出现在一个类或结构中时，`I` 必须表示那个类或结构的一个基本类的成员。同

样, 当 `base.[E]` 出现在一个类中时, 在其基类中必须存在一个可用的索引。在编辑期对式子 `base.I` 和 `base.[E]` 的 `base-access` 表达式进行求值就相当于把它们书写成 `((B)this).I` 和 `((B)this).[E]`, 这里, `B` 指的是发生此结构的类或结构的基本类。这样, 除 `this` 被看作这个基类的一个实异常, `base.I` 和 `base.[E]` 与 `this.I` 和 `this.[E]` 是一致的。

当 `base-access` 引用一个函数成员 (方法、属性或索引) 时, 为方便函数成员引用 (参见 7.4.3 节), 认为这个函数成员是非虚拟的。因此, 在一个 `virtual` 函数成员的 `override` 内, `base-access` 可被用来引用这个函数成员的继承工具。如果 `base-access` 引用的成员是抽象的, 则错误出现。

7.5.9 后缀增量和减量操作符 (Postfix Increment And Decrement Operators)

```
post-increment-expression:
primary-expression ++

post-decrement-expression:
primary-expression --
```

后缀增量或减量操作的操作数必须是作为变量、属性访问或索引访问的一个表达式。操作的结果是一与这个操作数类型相同的值。如果一个后缀增量或减量操作的操作数是--属性或索引访问, 则这个属性或索引必须具有 `get` 和 `set` 两个访问函数, 否则, 出现编辑期错误。

在选择一特定的操作符工具时应用一元操作符重载分解 (参见 7.2.3 节)。在下列类型中存在预先定义的操作符++和--: `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 及枚举类型。预先定义的操作符++的返回值是操作数加 1, 而--返回值为操作数减 1。

式子 `x++` 或 `x--` 的后缀增量或减量操作的运行期处理包括以下几步:

- (1) 如果 `x` 是一个变量, 则:
 - 对 `x` 求值产生这个变量
 - 保留 `x` 的值
 - 引用所选的操作符, `x` 的保留值作为其自变量
 - 将这个操作符的返回值存储在 `x` 的求值过程所给定的空间内
 - `x` 的保留值就是这个操作结果
- (2) 如果 `x` 是一个属性或索引访问:
 - 对实例表达式 (如果 `x` 不是 `static`) 以及与 `x` 有关的自变量列表 (如果 `x` 是一个索引访问) 进行求值, 且将其结果应用于下面的 `get` 和 `set` 访问函数的引用过程中。
 - 引用 `x` 的 `get` 访问函数并保留返回值
 - 引用所选的操作符, `x` 的保留值作为其自变量
 - 引用 `x` 的 `set` 访问函数, 由这个操作符所返回的值作为其 `value` 自变量
 - `x` 的保留值就是这个操作的结果

操作符++和--也支持前缀符号, 如 7.6.7 节所述。`X++` 或 `x--` 的结果就是操作前 `x` 的值, 而 `++x` 或 `--x` 的结果则是操作后 `x` 的值。在两种情况下, 操作后 `x` 本身的值不变。

用前缀或后缀符号可以引用 `operator` 工具++或--。两个符号不可能具有独立的工具操作符。

7.5.10 New 操作符 (New Operator)

new 操作符的作用是产生类型的 new 实例。

```
new-expression:
    object-creation-expression
    array-creation-expression
    delegate-creation-expression
```

new 表达式有三种形式:

- 对象建立表达式的作用是建立类类型和值类型的 new 实例。
- 数组建立表达式的作用是建立数组类型的 new 实例。
- 委托建立表达式的作用是建立委托类型的 new 实例。

new 操作符的含义是建立类型的一个实例,但不一定是指储存的动态分配。尤其是,值类型的实例除它们所在的变量外不需要额外的存储函数,且当 new 的作用是建立值类型的实例时,没有动态分配。

7.5.10.1 对象建立表达式 (Object Creation Expressions)

对象建立表达式(object-creation-expression)的作用是建立 class-type 或 value-type 的 new 实例。

```
object-creation-expression:
    new type {argument-listopt}
```

object-creation-expression 的类型必须是 class-type 或 value-type。这里的 type 不能是一个抽象的 class-type。

只有当 type 是 class-type 或 struct-type 时才允许有可选择的 argument-list(参见 7.4.1 节)。

在式子 $T(A)$ 中, T 是一个 class-type 或 value-type, A 是一个可选择的 argument-list, 则其 object-creation-expression 的编辑期处理包括以下几步:

- 如果 T 是一个 value-type, 而 A 不存在, 则:

这个 object-creation-expression 就是一个缺省的构造函数引用。其结果就是类型 T 的一个值, 称为 T 的缺省值, 如 4.1.1 节中所述。
- 否则, 如果 T 是一个 class-type 或 struct-type, 则:
 - 如果 T 是一个 abstract class-type, 则错误出现。
 - 要引用的构造函数通过应用 7.4.2 节中的重载分解规则来确定。候选构造函数的集合包括在 T 中声明的所有的可访问构造函数。如果候选构造函数的集合是空的, 或最佳构造函数不止一个, 则错误出现。
 - 此 object-creation-expression 的结果是类型 T 的一个值, 它是通过引用在以上步骤中所确定的构造函数而产生的。
- 否则, 这个 object-creation-expression 就是无效的, 错误出现。

式子 $\text{new } T(A)$ 的一个 object-creation-expression 的运行期处理包括以下几步, 其中 T 是一个 class-type 或 struct-type, A 是一个可选择的 argument-list:

- 如果 T 是一个 class-type:
 - 分配类 T 的一个 new 实例。如果没有足够可用的存储函数分配给这个 new 实例，则显示 `OutOfMemoryException`，且其它步骤也不再执行。
 - 把这个 new 实例的所有域初始化给它们的默认值（参见 5.2 节）。
 - 根据函数成员引用规则（参见 7.4.3 节）引用这个构造函数。对那个新分配的实例的引用被自动传递给这个构造函数，且这个实例在 `this` 这样的构造函数中可访问。
- 如果 T 是一个 struct-type:
 - 通过指定一个临时局部变量建立类型 T 的一个实例。由于需要 struct-type 的构造函数给每一个正在建立的实例的域明确赋一个值，因此，这个临时变量不必初始化。
 - 根据函数成员引用规则（参见 7.4.3 节）引用构造函数。对新分配的实例的引用被自动传递给那个构造函数，且这个实例在 `this` 这样的构造函数中可访问。

7.5.10.2 数组建立表达式 (Array Creation Expressions)

数组建立表达式 (array-creation-expression) 作用是建立 array-type 的一个 new 实例。

```
array-creation-expression:
    new          non-array-type [expression-list]          rank-specifiersCp1
    array-initializeropt
    new array-type array-initializer
```

第一种形式的数组建立表达式指定一类型的数组实例，这个类型是通过从表达式列表中删除每一个独立的表达式而得到的。例如，数组建立表达式 `new int[10,20]` 产生类型 `int[,]` 的一个数组实例，而数组建立表达式 `new int[10][,]` 产生类型 `int[][,]` 的一个数组实例。表达式列表中的每一个表达式的类型必须是 `int`、`uint`、`long`、`ulong` 或通过隐式转换可转化为这些类型中的一个或多个的类型。每一个表达式的值都决定新分配的数组实例中相应的范围长度。

如果第一种形式的一个数组建立表达式包括一个数组初始化，则表达式列表中的每一个表达式都必须是一个常量，且由自变量列表所规定的它们的等级和范围长度必须同这个数组初始化的相等。

在第二种形式的数组建立表达式中，指定数组类型的等级必须同那个数组初始化的相等。单独的范围长度可以从这个数组初始化的每一个嵌套水平的元素数来推断。因此，表达式

```
new int[, ] {{0,1},{2,3},{4,5}};
```

与下面的表达式一致：

```
New int [3,2] {{0,1},{2,3},{4,5}}
```

关于数组初始化将在 12.6 节进一步阐述。

对数组建立表达式求值的结果是一个值，称为对新分配的数组实例的一个引用。数组建立表达式的运行期处理包括以下几步：

- 对 expression-list 的范围长度表达式从左到右按顺序求值。对每一个表达式求值之后，执行一个到类型 int 的隐式转换（参见 6.1 节）。如果一个表达式的求值过程或其后的隐式转换出现异常，则其它的表达式也不再被求值，执行停止。
- 检验范围长度计算值的有效性。如果有一个或多个这样的值比零小，则显示 `IndexOutOfRangeException`，且执行停止。
- 分配一个给定范围长度的数组实例。如果没有足够可用的存储函数分配给这个 new 实例，则显示 `OutOfMemoryException`，且执行停止。
- 把新数组实例的所有元素初始化给它们的缺省值（参见 5.2 节）。
- 如果此数组建立表达式包括一数组初始化，则对这个数组初始化中的每一个表达式进行求值，并把这个值赋给其相应的数组元素。求值和赋值过程都按照数组初始化中这些表达式的书写顺序来执行。也就是说，元素是以升序索引而被初始化的，最右边的范围首先增加。

如果一给定表达式的求值过程或其后相应数组元素的赋值过程出现异常，则其它元素也不再被初始化（且剩下的元素将因此而获得它们的缺省值）。

数组建立表达式允许具有数组类型元素的数组的示例（用例子具体说明），但是，这样一个数组的元素必须被人为地初始化。例如，语句

```
int[,] a = new int[100][];
```

产生一个具有 100 个类型为 `int[]` 的元素的单维数组，每一个元素的初始值都是 `null`。数组建立表达式的子数组不可能相同，且语句

```
int[,] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

是错误的。子数组的示例必须被人为执行，如下面的例子：

```
int[,] = new int[100, 5];
```

当数组群中的一个数组呈矩形时，也就是说，当子数组的长度相等时，用多维数组效率更高。在上面的例子中，数组群中该数组的示例产生了 101 个对象——其中一个外部数组，100 个子数组。而

```
int[,] = new int[100, 5];
```

只产生一个对象，即一个二维数组，并完成在一个语句中的分配。

7.5.10.3 委托建立表达式 (Delegate Creation Expressions)

委托建立表达式 (delegate-creation-expression) 的作用是建立 `delegate-type` 的一个 new 实例。

```
delegate-creation-expression:
    new delegate-type { expression }
```

委托建立表达式的自变量必须是一个方法群或 `delegate-type` 的一个值。如果这样的自变

量是一个方法群，则它识别其方法。对于一个实例方法来说，它识别要建立委托的对象。如果自变量是 `delegate-type` 的一个值，则它识别要建立拷贝的委托实例。

在式子 `D(E)` 中，`D` 是一个 `delegate-type`，`E` 是一个 `expression`，它的一个 `delegate-creation-expression` 编辑期处理包括以下几步：

- 如果 `E` 是一个方法群，则：
 - 如果这个方法群是由 `base-access` 得来的，则错误出现。
 - 被 `E` 所识别的方法的集合必须只包括一个方法，这个方法具有同样的签名且返回类型与 `D` 的相同，即新产生的委托所指的那个方法。如果不存在相匹配的方法，或这样的方法多于一个，则错误出现。如果被选择的方法是一个实例方法，与 `E` 有关的实例表达式就决定委托的目标对象。
 - 与在方法引用中相同，所选择的方法必须适合于此方法群的上下文：
 - 如果这个方法是一个静态方法，则这个方法群一定是按照类型从 `simple-name` 或 `member-access` 中得来的。如果这个方法是一个实例方法，则这个方法群必须是按照变量或值从 `simple-name` 或 `member-access` 中得到的。如果所选择的方法与上下文不符，则错误出现。
 - 其结果是类型 `D` 的一个值，称为一个新建立的委托，这里指的是被选择的方法或目标对象。
- 否则，如果 `E` 是 `delegate-type` 的一个值，则：
 - `E` 的 `delegate-type` 必须具有绝对相同的签名，且返回类型为 `D`，否则，错误出现。
 - 结果为类型 `D` 的一个值，称为一个新建立的委托，这里指的是相同的方法及象 `E` 这样的目标对象。
- 否则，这个委托建立表达式就是无效的，错误出现。

在 `newD(E)` 中，`D` 是一个 `delegate-type`，`E` 是一个 `expression`，其 `delegate-creation-expression` 的运行期处理包括以下几步：

- 如果 `E` 是一个方法群，则：
 - 如果在编辑期所选择的方法是一个静态方法，则这个委托的目标对象就是 `NULL`。否则，所选择的方法就是一个实例方法，且此委托的目标对象是从与其有关的实例表达式中确定的：

对这个实例表达式进行求值。如果求值出现异常，则其它步骤也停止执行。如果这个实例表达式是一个 `reference-type`，则由这个实例表达式所计算的值就是目标对象。如果这个目标对象是 `null`，则显示 `NullReference Exception`，执行停止。

如果这个实例表达式是一个 `value-type`，则执行一个封箱操作（参见 4.3 节）把此值转化为一个对象，这个对象即目标对象。

- 分配委托类型 `D` 的一个 `new` 实例。如果没有足够可用的存储单元分配给这个 `new` 实例，则显示 `Out of Memory Exception`，执行停止。
- 用编辑期所确定的方法及上面所计算的目标对象的引用初始化这个 `new` 委托实例。

- 如果 E 是 delegate-type 的一个值。
 - 对 E 赋值, 如果此赋值过程出现异常, 则下面的步骤将不再执行。
 - 如果 E 的值是 null, 则显示 NullReference Exception, 执行停止。
 - 对委托类型 D 的一个 new 实例进行分配。如果没有足够可用的存储单元分配给这个 new 实例, 则显示 OutOfMemory Exception, 其它步骤也不再执行。
 - 用与由 E 给定的委托实例具有相同的方法和对象的引用初始化这个 new 委托实例。

一个委托所指的方法和对象是在这个委托被实例时确定的, 并在这个委托存在的整个时期内保持不变。也就是说, 委托的目标方法或对象一旦建立, 就很难改变。

不可能建立这样一个委托, 即表示构造函数、属性、索引或用户自定义操作符的委托。

如上所述, 当一个委托是在一方法群中建立时, 其签名和返回类型就决定了要选择哪一个重载方法。例如:

```
delegate double DoubleFunc(double x);

class A
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }

    static double Square(double x) {
        return x * x;
    }
}
```

域 A.F 被一委托初始化, 这个委托指的是第二个 square 方法, 因为该方法与 koublefunc 的签名及返回类型完全符合。如果第二个 square 方法不存在, 则出现编辑期错误。

7.5.11 Typeof 操作符 (Type of Operator)

typeof 操作符的作用是取得类型的对象 system.type。

```
typeof-expression:
    typeof ( type )
```

一个 typeof expression 的结果是所示类型的 system.type 对象。

例如:

```
class Test
{
    static void Main() {
```

```

Type[] t = {
    typeof(int),
    typeof(System.Int32),
    typeof(string),
    typeof(double[])
};

for (int i = 0; i < t.Length; i++) {
    Console.WriteLine(t[i].Name);
}
}

```

输出的结果为:

```

Int32
Int32
String
Double[]

```

注意: int 和 system.int32 的类型相同。

7.5.12 检查的和未检查操作符 (Checked And Unchecked Operators)

检查和未检查操作符的作用是控制整类型算术操作和转换的 overflow checking context。

```

checked-expression:
    checked { expression }

unchecked-expression:
    unchecked { expression }

```

checked 操作符对已检查上下文中所包含的表达式进行求值, 而 unchecked 则对未检查的上下文中所包含的表达式进行求值。除被包含的表达式在所给定的溢出检查上下文中被求值外, checked-expression 或 unchecked-expression 与 parenthesized-expression (参见 7.5.3 节) 完全一致。

溢出检查上下文也只能通过 checked 和 unchecked 语句 (参见 8.11 节) 被控制。

由 checked 和 unchecked 操作符和语句所确定的溢出检查上下文影响下列操作:

- 预定义的一元操作符 (参见 7.5.9 节和 7.6.7 节) ++ 和 --, 当其操作数为整类型时。
 - 预定义的一元操作符 (参见 7.6.2 节) -, 当其操作数为整类型时。
 - 预定义的算术运算符 (参见 7.7 节) +、-、* 和 /, 当两个操作数都是整类型时。
 - 从一整类型到另一整类型的显式数值转换 (参见 6.2.1 节)。
- 当上面的操作中有一个产生的结果太大而不能表示其类型时, 执行那个操作的上下文就控制结果:
- 在 checked 上下文中, 如果操作是一个常量表达式 (参见 7.1.5 节), 则出现编辑期错误。否则, 当操作是在运行期执行时, 显示 overflow exception。

- 在 `unchecked` 上下文中，结果被缩短，丢弃了那些不适合目的类型的高位。

当一个非常量表达式（在运行期被求值的表达式）不包括在任何 `checked` 或 `unchecked` 操作符或语句之内时，溢出在这个表达式的运行期求值过程中的影响取决于外部因素（比如编辑人员的替换及执行环境的结构）。然而，这种影响一定要么是 `checked` 求值过程的影响，要么是 `unchecked` 求值过程的影响。

对于常量表达式（在编辑期可被完全求值的表达式）来说，缺省的溢出检查上下文总是 `checked`，除非一个常量表达式在 `unchecked` 上下文中被显式替换，否则，在这个表达式的编辑期求值过程中出现的溢出总是导致编辑期错误。在下面的例子中不存在编辑期错误，因为两个表达式都不能在编辑期被求值：

```
class Test
{
    static int x = 1000000;
    static int y = 1000000;

    static int F() {
        return checked(x * y);    // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y); // Returns -727379968
    }

    static int H() {
        return x * y;             // Depends on default
    }
}
```

在运行期，方法 `F()` 显示一个 `overflow exception`，而方法 `g()` 返回—727379968（超范围结果较低的 32 位）。方法 `H()` 的运行情况取决于编辑缺省的溢出检查上下文，但它总是与 `F()` 或 `G()` 相同。在下面的例子中：

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;
    static int F() {
        return checked(x * y);    // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y); // Returns -727379968
    }
}
```



```

static int H() {
    return x * y;           // Compile error, overflow
}
}

```

发生在对 `f()` 和 `h()` 中常量表达式的求值过程中的溢出导致编辑期错误，因为表达式在 `checked` 上下文中被求值。当对 `g()` 中的常量表达式求值时，也会出现溢出，但由于求值发生在 `unchecked` 上下文中，因此，不显示溢出。

`checked` 和 `unchecked` 操作符只影响那些原文包括在符号 “[” 和 “]” 之内的操作的溢出检查上下文。这些操作符不影响这样的函数成员，即被作为对所含表达式求值的一个结果而引用的函数成员。在下面的例子中 `F()` 中 `checked` 的应用并不影响 `multiply()` 中 `x*y` 的求值，因此 `x*y` 在缺省溢出检查上下文中被求值：

```

class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}

```

当在六进制中书写带符号的整类型常量时，使 `unchecked` 操作符更方便。例如：

```

class test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}

```

上面两个六进制常量的类型都是 `uint`。因为这些常量在 `int` 范围之外，没有 `unchecked` 操作符，对 `int` 的计算产生编辑期错误。

7.6 一元表达式 (Unary Expression)

一元表达式中涉及的操作符如下：

```

unary-expression:
primary-expression
+ unary-expression
- unary-expression

```

```
! unary-expression
~ unary-expression
* unary-expression
& unary-expression
pre-increment-expression
pre-decrement-expression
cast-expression
```

7.6.1 一元加操作符 (Unary Plus Operator)

对于形式+X的操作来说,用一元操作符重载分解(参见7.2.3节)来选择特定的操作符执行。操作数被转化为所选操作符的参数类型,且结果类型就是这个操作符的返回类型。预定义的+一元操作符是:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

对于每一个这些操作符而言,其结果只是操作数的值。

7.6.2 一元减操作符 (Unary Minus Operator)

对于形式-X的一个操作来说,一元操作符重载分解的作用是选择一个特定的操作符执行(执行)。操作数被转化为所选操作符的参数类型,且结果类型就是这个操作符的返回类型。预定义否定操作符是:

- 整体否定

```
int operator -(int x);
long operator -(long x);
```

其结果是通过0减去X而得到的。在一个checked上下文中,如果x的值是最大的否定int或long,则显式overflow exception。在一个unchecked上下文中,如果x的值是最大的否定int或long,其结果就是那个值,且不显示溢出。

如果否定操作符的操作数是类型uint,则它被转化为类型long,且结果类型就是long。但有一种特殊情况异常,即允许把int值-2147483648(-2的31次幂)作为一个十进制整型字母(参见2.4.4.2节)来书写。

如果这个否定操作符的操作数类型是 `ulong`，则错误出现。但下列情况异常，即允许把 `long` 值 -9223372036854775808（-2 的 63 次幂）作为一个十进制整型字母来书写。

- 浮点数否定

```
float operator -(float x);
double operator -(double x);
```

其结果就是改变符号 `X` 的值。如果 `X` 是 `nan`，则结果也是 `nan`。

- 十进制否定

```
decimd operator -(decimd x);
```

其结果是用 0 减去 `X` 而得到的。

7.6.3 逻辑非操作符 (Logical Negation Operator)

对于式子 `!X` 的操作来说，一元操作符重载分解的作用是选择一特定的操作符工具。这个操作数被转化为所选操作符的参数类型，且结果类型就是这个操作符的返回类型。只有一个预定义的逻辑非操作符：

```
bool operator!(bool x);
```

这个操作符计算此操作数的逻辑非。如果这个操作数是 `true`，则结果就是 `false`。如果这个操作数是 `false`，则结果是 `true`。

7.6.4 按位求补码操作符 (Bitwise Complement Operator)

对于式子 `~X` 的操作来说，一元操作符重载分解的作用是选择一特定的操作符工具。这个操作数被转化为所选操作符的参数类型，且结果类型就是此操作符的返回类型。预定义的按位求补码操作符是：

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

对于这些操作符的每一个来说，此操作的结果都是 `X` 的按位求补。

每一个枚举类型 `E` 都固定提供下面的按位执行操作符：

```
e operator ~(e, x);
```

对 `~X` 求值的结果与对 `(E) (~ (U) X)` 求值的结果相同，这里 `X` 是带有隐藏类型 `U` 的枚举类型 `E` 的一个表达式。

7.6.5 前缀增量和减量操作符 (Prefix Increment And Decrement Operators)

```
pre-increment-expression:
    ++ unary-expression

pre-decrement-expression:
    -- unary-expression
```

前缀增量或减量操作的操作数必须是变量、属性访问或索引访问的表达式。其操作的结果就是与该操作数类型相同的一个值。

如果一个前缀增量或减量操作的操作数是一个属性或索引访问，那么，这个属性或索引必须既有 get 访问函数，又有 set 访问函数。否则，编辑时将出现错误。

一元操作符重载分解的作用是选择一特定的操作符工具。预定义操作符++和--有以下几种类型：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal 和任何枚举类型。预定义操作符++的返回值是原操作数加 1，预定义操作符--的返回值为原操作数减 1。

式子 ++x 或 --x 的前缀增量或减量操作的运行期处理包括以下几步：

- 如果 x 是一个变量：
 - 对 x 求值产生这个变量。
 - 引用所选择的操作符，x 的值作为其自变量。
 - 把这个操作符的返回值存储在 x 的求值过程所给定的空间。
 - 此操作符的返回值就是这个操作的结果。
- 如果 x 是一个属性或索引访问：
 - 对与 x 有关的实例表达式（如果 x 不是 static）和自变量列表（如果 x 是一个索引访问）进行求值，并将其结果应用于其后的 get 和 set 访问函数引用过程中。
 - 引用 x 的 get 访问函数。
 - 引用所选的操作符，访问函数 get 的返回值作为其自变量。
 - 引用 x 的 set 访问函数，此操作符的返回值作为其 value 自变量。
 - 此操作符的返回值就是这个操作的结果。

操作符++和--也支持后缀符号，如 7.5.9 节所述。x++或 x--的结果就是操作前的 x 值，而 ++x 或 --x 的结果都为操作后 x 的值。两种情下，操作后 x 本身的值都不变。

用后缀或前缀符号可引用一个 operator++或 operator--工具。两种符号不能有独立的操作符功能。

7.6.6 CAST 表达式 (Cast Expressions)

cast-expression 的作用是把一个表达式显式转化为一给定类型。

```
cast-expression:
    (type) unary-expression
```

式子 (T) E 的 cast-expression 执行一个从 E 的值到类型 T 的显式转换（参见 6.2 节），

其中, T 是一个 `type`, E 是一个 `unary-expression`。如果不存在从 E 的类型到 T 的显式转换, 则错误出现。否则, 其结果就是由这个显式转换产生的一个值。即使 E 表示一个变量, 结果也是一个值。

`cast-expression` 的语法使得某些句法意义不明确。例如, 表达式 $(X) - Y$ 既可被解释为一个 `cast-expression` (从 Y 到类型 X 的一个引用), 也可被解释为一个与 `parenthesized-expression` (它计算 $X - Y$ 的值) 联合的 `additive-expression`。

为解决 `cast-expression` 的意义不明确问题, 有以下规则, 只有当至少下列条件之一符合时, 括弧内所包含的一个或多个 `tokens` (参见 2.4 节) 的序列才被认为是一个 `cast-expression` 的开始:

- 符号的序列对于 `type` 来说语法是正确的, 而对于 `expression` 来说则是不正确的。
- 符号的序列对于 `type` 来说语法是正确的, 且紧接着在其括弧后面的符号是 “~”、“!”、“(”、一个 `identifier` (参见 2.4.1 节)、字母 (参见 2.4.4 节) 或除 `is` 之外的任何 `keyword` (参见 2.4.3 节)。

上面的规则意思就是, 只有当常量明显是一个 `cast-expression` 时, 才能认为它是一个 `cast-expression`。

上面的术语“正确语法”只是指, 符号的顺序必须与特定的语法产生过程一致。但它并不涉及任何形式标识符的实际意义。例如, 如果 X 和 Y 是标识符, 那么, 即使 X, Y 实际并不表示一个类型, 对于类型来说, X, Y 的语法也是正确的。

由上面这些明确的规则可以得出, 如果 X 和 Y 都是标识符, 即使 X 识别一个类型, $(X) Y$ 、 $(X) (Y)$ 以及 $(X) (-Y)$ 都是 `cast-expressions`, 但 $(X) - Y$ 不是。然而, 如果 X 是一个关键词, 这个关键词标识一个预定义类型 (比如 `int`), 那么, 所有四种形式都是 `cast-expressions` (因为这样一个关键词本身不能是一个表达式)。

7.7 算术运算符 (Arithmetic Operators)

操作符 `*`、`/`、`%`、`+` 和 `-` 称为算术运算符。

```

multiplicative-expression:
    unary-expression
multiplicative-expression * unary-expression
multiplicative-expression / unary-expression
multiplicative-expression % unary-expression

additive-expression:
    multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

```

7.7.1 乘法运算操作符 (Multiplication Operator)

对于式子 $X * Y$ 的一个操作符来说, 二进制操作符重载分解 (参见 7.2.4 节) 的作用是选择特定的操作符功能。这些操作数被转化为所选操作符的参数类型, 且结果类型就是这个操作符的返回类型。

预定义乘法运算操作符列表如下。这些操作都是 X 和 Y 的积。

- 整型乘法:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

在一 checked 上下文中, 如果这个积在结果类型的取值范围之外, 则显示 overflow exception。在一 unchecked 上下文中, 显示溢出且舍弃这个结果的任何有意义的高位。

- 浮点乘法:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

这个积的运算依据 IEEE754 算术的规则。下面的表格列举了非零有穷值、零、无穷值及 NaN 的所有可能结合方式的结果。在表 7-3 中, X 和 Y 都是正有穷限值。Z 是 X*Y 的结果。如果结果对于目的类型来说太大, 则 Z 就是无穷。如果结果对于目的类型来说太小, 则 Z 就是零。

表 7-3 积的运算

	-y	y	+0	-0	-∞	-∞	NaN
+x	z	-z	+0	-0	-∞	∞	NaN
-x	-z	z	0	+0	∞	+∞	NaN
+0	+0	0	+0	-0	NaN	NaN	NaN
-0	0	+0	-0	+0	NaN	NaN	NaN
+∞	+∞	∞	NaN	NaN	+∞	-∞	NaN
-∞	-∞	-∞	NaN	NaN	-∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

```
decimal operator *(decimal x, decimal y);
```

如果结果值太大而不能以 decimal 格式来表示, 则显示 overflow exception。如果结果值太小不能以 decimal 格式来表示, 则结果就是零。

7.7.2 除法运算操作符 (Division operator)

对于式子 X/Y 的操作来说, 二进制操作符重载分解 (参见 7.2.4) 的作用是选择一特定的操作符执行。操作数被转化为所选操作符的参数类型, 且结果类型就是这个操作符的返回类型。

预定义除法操作符列表如下。这些操作符都计算 X 与 Y 的商。

- 整型除法:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
```

```
long operator / (long x, long y);
ulong operator / (ulong x, ulong y);
```

如果操作数的值是零，则显示 divide by zero exception。

除法的结果可以四舍五入到零位，且结果的绝对值就是比这两个操作数的商小的最大可能的整数。当这两个操作数具有相同的符号时，其结果为 0 或正数，而当这两个操作数具有相反的符号时，其结果为 0 或负数。如果左操作数是最大的负 int 或 long，而右操作数是 -1，则发生溢出。在一 checked 上下文中，这会导致显示 overflow exception。在一 unchecked 上下文中，不显示溢出，其结果就是左操作数的值。

● 浮点数除法：

```
float operator / (float x, float y);
double operator / (double x, double y);
```

商是根据 IEEE754 算术规则算出的。表 7-4 列举了非零有穷值、零、无穷及 NaN 的所有可能结合方式的结果。在这个表中，X 和 Y 都是正有穷值。Z 是 X/Y 的结果。如果结果对于目的类型来说太大，则 Z 的值是无穷。如果结果对于目的类型来说太小，则 Z 的值就是零。

表 7-4 商的运算

	+y	-y	+0	-0	+∞	-∞	NaN
+x	z	-z	+∞	-∞	+0	-0	NaN
-x	-z	z	-∞	+∞	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+∞	+∞	-∞	+∞	-∞	NaN	NaN	NaN
-∞	-∞	+∞	-∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

● 十进制除法：

```
decimal operator / (decimal x, decimal y);
```

如果右操作数的值是零，则显示 divide by zero exception。如果结果值太大而不能以 decimal 格式来表示，则显示 overflow-exception。如果结果值太小，不能以 decimal 格式来表示，则结果为零。

7.7.3 求余数操作符 (Remainder Operator)

对于式子 $X \% Y$ 的操作来说，二进制操作符重载分解（参见 7.2.4 节）的作用是选择特定的操作符执行。这些操作数被转化为所选操作符的参数类型，且结果类型就是这个操作符的返回类型。

预定义求余操作符列表如下。这些操作符都用来求 X 与 Y 相除的余数。

- 整型求余：

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

$X\%Y$ 的结果就是由 $X-(X/Y)*Y$ 产生的值。如果 Y 是零，则显示 divide by zero exception。求余操作绝不会导致溢出。

- 浮点数求余：

```
float operator %(float x, float y);
double operator %(double x, double y);
```

表 7-5 列举了非零有穷值、零、无穷以及 nan 的所有可能结合方式的结果。在这个表中， X 和 Y 都是正有穷值。 Z 是 $X\%Y$ 的最大可能的整数。这种求余方法与在整型操作数中所用的方法相似，但与 IEEE754 中的定义（其中， N 是与 X/Y 最接近的整数）不同。

表 7-5 求余操作

	$+\infty$	$-\infty$	$+0$	-0	$+\infty$	$-\infty$	NaN
$+\infty$	Z	Z	NaN	NaN	x	x	NaN
$-\infty$	$-Z$	$-Z$	NaN	NaN	$-x$	$-x$	NaN
$+0$	$+0$	$+0$	NaN	NaN	$+0$	$+0$	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
$+\infty$	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$-\infty$	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 十进制求余：

```
decimal operator %(decimal x, decimal y);
```

如果右操作数的值是零，则显示 divide by zero exception。如果结果值太大而不能以 decimal 格式来表示，则显示 overflow exception。如果所得值太小，不能以 decimal 格式来表示，则结果为零。

7.7.4 加法操作符(Addition Operator)

对于式子 $X+Y$ 来说，二进制操作符重载分解（参见 7.2.4 节）的作用是选择特定的操作符执行。操作数被转化为所选操作符的参数类型，且结果类型就是操作符的返回类型。

预定义加法操作符列表如下。对于数值及枚举类型来说，预定义加法操作符计算两个操作数的和。当有一或两个操作数都是字符串类型时，预定义的加法操作符把这些字符串连在

一起（中间没有空格）。

- 整数加法：

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

在 checked 上下文中，如果所得和在结果类型的取值范围之外，则显示 overflow exception。在一 unchecked 上下文中，不显示溢出，且舍弃这个结果任何有意义的高位。

- 浮点数加法：

```
float operator +(float x, float y);
double operator +(double x, double y);
```

这个和是依据 IEEE754 算术规则而算出的。下面的表格列出了非零有穷值、零、无穷以及 NaN 的所有可能结合方式的结果。表 7-6 中，X 和 Y 都是非零有穷值，Z 是 X+Y 的结果。如果 X 和 Y 大小相等但符号相反，则 Z 的值就是正零。如果 X+Y 太大而不能以目的类型来表示，则 Z 就是与 X+Y 符号相同的无穷。如果 X+Y 太小而不能以目的类型来表示，则 Z 就是与 X+Y 符号相同的零。

表 7-6 加法操作

	y	+0	-0	+∞	-∞	NaN
x	z	x	x	+∞	-∞	NaN
+0	y	+0	+0	+∞	-∞	NaN
-0	y	+0	-0	+∞	-∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
-∞	-∞	-∞	-∞	NaN	-∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 十进制加法：

```
decimal operator +(decimal x, decimal y);
```

如果结果值太大而不能以 decimal 格式来表示，则显示 overflow exception。如果结果值太小而不能以 decimal 格式来表示，则其结果就是零。

- 枚举加法。每一个枚举类型固定提供下列预定义操作符，这里，E 就是这个枚举类型，U 是 E 的潜在类型：

```
E operator +(E X, U Y);
E operator +(U X, E Y);
```

这些操作符的求值同 (E)((U) X + (U) Y)。

- 字符串连接:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

当有一或两个操作数的类型都是 `string` 时, 二进制操作符+执行字符串连接。如果字符串连接的一个操作数是 `null`, 则用一个空串来表示。否则, 任何非字符串自变量都要转化为其字符串表示形式, 这个转化可以通过引用从类型 `object` 继承而来的虚拟方法 `ToString()` 来完成。如果 `ToString()` 返回 `null`, 则一个空串被取代。

字符串连接操作符的结果是一个字符串, 这个字符串包括左操作数的字母及其后的右操作数的字母。字符串连接操作符绝不会产生一个 `null` 值。如果没有足够可用的存储单元分配给所得的字符串, 则显示 `OutOfMemoryException`。

- 委托结合。每一个委托类型都固定提供下列预定义操作符, 这里, `D` 就是这个委托类型: `D operator +(D X, D Y);`

当有一或两个操作数都是委托类型 `D` 时, 二进制操作符+执行委托结合。如果第一个操作数是 `null`, 那么, 操作的结果就是第二个操作数。否则, 如果第二个操作数是 `null`, 那么, 操作的结果就是第一个操作数的值。否则, 如果 `D` 是一个可组合的委托类型(参见 15.1.1 节), 则操作的结果就是一个 `new` 委托实例, 当此实例被引用时, 首先引用第一个操作数, 然后是第二个。否则, `D` 就是一个不可组合的委托类型, 显示 `MulticastNotSupportedException`。

7.7.5 减法操作符 (Subtraction Operator)

对于式子 `X-Y` 的一个操作来说, 二进制操作符重载分解(参见 7.2.4 节)的作用是选择一特定的操作符执行。操作数被转化为所选操作符的参数类型, 且结果类型就是这个操作符的返回类型。

预定义的减法操作符列表如下。这些操作符都执行 `X` 减去 `Y` 的运算。

- 整数减法:

```
int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);
```

在一 `checked` 上下文中, 如果这个差值在结果类型的取值范围之外, 则显示 `OverflowException`。在一 `unchecked` 上下文中, 不显示溢出, 并舍弃这个结果任何有意义的高数位。

- 浮点数减法:

```
float operator -(float x, float y);
double operator -(double x, double y);
```

这个差可依据 IEEE754 算术规则来算出。下面的表格列出了非零有穷值、零、无穷以及 `NAN` 的所有可能结合方式的结果。表 7-7 中, `X` 和 `Y` 都是非零有穷值, `Z` 是 `X-Y` 的结果。

如果 X 和 Y 相等, 则 Z 就是正零。如果 $X-Y$ 太大而不能以目的类型来表示, 则 Z 就是一个与 $X-Y$ 符号相同的无穷。如果 $X-Y$ 太小而不能以目的类型来表示, 则 Z 就是与 $X-Y$ 符号相同的零。

表 7-7 差的操作

	y	+0	-0	+∞	-∞	NaN
x	z	x	x	-∞	+∞	NaN
+0	-y	+0	+0	-∞	+∞	NaN
-0	-y	-0	-0	-∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
-∞	-∞	-∞	-∞	-∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 十进制减法:

`decimal operator - (decimal x, decimal y);`

如果结果值太大而不能以 `decimal` 格式来表示, 则显示 `overflow exception`。如果结果值太小, 不能以 `decimal` 格式来表示, 则结果为零。

- 枚举减法。每一个枚举类型都固定提供下列预定义操作符, 这里, E 就是这个枚举类型, U 是 E 的潜在类型: `U operator - (EX, EY);`

这个操作符的求值与 $(U)(U) X - (U) Y$ 相同。也就是说, 此操作符计算 X 和 Y 的顺序差值, 且结果类型就是这个枚举的潜在类型: `E operator - (EX, EY);`

此操作符的求值与 $(E)(U) X - Y$ 相同。也就是说, 这个操作符从枚举的潜在类型减去一个值, 得到这个枚举的值。

- 委托排除。每一个委托类型都固定提供下列预定义操作符, 这里, D 是一个委托类型: `D operator - (DX, DY);`

当有一或两个操作数都是委托类型 D 时, 二进制操作符—执行委托排除。

7.8 转换操作符 (Shift Operators)

操作符 `<<` 和 `>>` 的作用是进行位转换操作。

```

shift-expression:
additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression

```

对于式子 $X \ll \text{count}$ 或 $X \gg \text{count}$ 的操作来说, 二进制操作符重载分解的作用是选择一特定的操作符执行。操作数被转化为所选操作符的参数类型, 且结果类型就是这个操作符的返回类型。

当声明一个重载转换操作符时，第一个操作数的类型必须包括这个操作符声明的类或结构，且第二个操作数的类型必须是 `int`。

预定义转换操作符列表如下。

- 左转换：

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

操作符 `<<` 转换下面所运算的位左边的 `X`：`X` 的高序位被删除，剩下的位被左转换，低序空位以 0 来补充。

- 右转换：

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

操作符 `>>` 转换下面所运算的位右边的 `X`。当 `X` 是类型 `int` 或 `long` 时，删去低序位，剩下的位被右转换，如果 `X` 是非负值，则高序空位以零来补充，如果 `X` 是负值，则以 1 来补充。当 `X` 是类型 `uint` 或 `ulong` 时，删除 `X` 的低序位，剩下的位被右转换，高序空位以零来补充。

对于预定义操作符来说，要转换的位数计算方法如下：

- 当 `X` 的类型是 `int` 或 `uint` 时，转换数由 `count` 的低序五位决定。也就是说，这个转换数是从 `count&0x1f` 计算得来的。
- 当 `X` 的类型是 `long` 或 `ulong` 时，转换数是由 `count` 的低序六位决定。也就是说，转换数是通过计算 `count&0x3f` 而得来的。

如果所得的转换数是 0，则这些转换操作符只返回 `X` 的值。转换操作绝不会导致溢出及在 `checked` 和 `unchecked` 上下文中产生相同的结果。当操作符 `>>` 的左操作数是一带符号的整类型时，这个操作符就执行一个 `arithmetic` 右转换，把这个操作数最有意义的位（带符号的位）的值传给高序空位。当操作符 `>>` 的左操作数是一个不带符号的整类型时，这个操作符就执行一个 `logical` 右转换，其中，高序空位总被设置为 0。为了执行与据操作数类型的推断结果相反的操作，可以用显式计算。例如，如果 `X` 是类型 `int` 的变量，那么，操作 `(int)((uint)X) >> Y` 执行 `X` 的逻辑右转换。

7.9 关系操作符 (Relational Operators)

操作符 `=`、`!=`、`<`、`>`、`<=`、`>=`、`is` 和 `as` 称为关系操作符。

```
relational-expression:
    shift-expression
relational-expression < shift-expression
```

```

relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
relational-expression is type
relational-expression as type

equality-expression:

relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

```

关于 is 操作符的说明见 7.9.9 节，关于 as 操作符的说明见 7.9.10 节。

操作符 ==、!=、<、>、<= 和 >= 是 comparison operators。对于式子 XopY 的操作来说，重载分解的作用是选择一特定的操作符执行，这里，op 是一个比较操作符。操作数被转化为所选操作符的参数类型，且结果类型就是所选操作的返回类型。

预定义的比较操作符将在以下几部分加以描述。所有预先确定的比较操作符都返回一个类型为 bool 的结果，如表 7-8 所述。

表 7-8 比较操作符的操作

操 作	结 果
x == y	如果 x 等于 y，则为 true；否则为 false
x != y	如果 x 不等于 y，则为 true；否则为 false
x < y	如果 x 小于 y，则为 true；否则为 false
x > y	如果 x 大于 y，则为 true；否则为 false
x <= y	如果 x 小于等于 y，则为 true；否则为 false
x >= y	如果 x 大于等于 y，则为 true；否则为 false

7.9.1 整数比较操作符 (Integer Comparison Operators)

预定义整数比较操作符是：

```

bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);
bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

```

```
bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

这些操作符的每一个都比较两个整类型操作数的数值，而返回一个 bool 值，这个 bool 值表明其关系是 true 还是 false。

7.9.2 浮点数比较操作符 (Floating-Point Comparison Operators)

预定义浮点数比较操作符是：

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);
```

```
bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

这些操作符根据 IEEE754 标准规则比较操作数:

- 如果两个操作数有一个是 NAN, 则对除 != 之外的所有操作符来说, 结果都是 FALSE, 而对 != 来说, 结果为 true。对任何两个操作数来说, $X != Y$ 总是产生与 $!(X=Y)$ 相同的结果。然而, 当有一个或两个操作数都是 NAN 时, 操作符 <, <= 和 >= 不产生与相反操作符的逻辑非相同的结果。例如, 如果 X 和 Y 有一个是 NAN, 那么 $X < Y$ 就是 FALSE, 但 $!(X >= Y)$ 是 TRUE。
- 当两个操作数都不是 NAN 时, 这些操作符按顺序比较两个浮点操作数的值: -无穷大 < -MAX<...< -MIN< 0.0 < +0.0 < +MIN<...< +MAX< +无穷大。这里, MIN 和 MAX 分别是可用给定浮点格式表示的最小和最大的正有穷值。由此顺序可以得出:
 - 正零和负零相等。
 - 一个负无穷总是小于其它值, 而等于另一个负无穷。
 - 一个正无穷总是大于其它值, 而等于另一个正无穷。

7.9.3 十进制比较操作符 (Decimal Comparison Operators)

预定义十进制比较操作符是:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

这些操作符中的每一个都比较两个十进制操作数的数值, 而返回一个 bool 值, 这个 bool 值表明其关系是 true 还是 false。

7.9.4 布尔等操作符 (Boolean Equality Operators)

预定义的布尔等操作符是:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

如果 X 和 Y 都是 true 或都是 false, 则 == 的结果是 true。否则, 其结果就是 false。

如果 X 和 Y 都是 true 或都是 false, 则 != 的结果是 false。否则, 其结果就是 true。当这些操作数是 bool 型时, 操作符 != 与操作符 ^ 的结果相同。

7.9.5 枚举比较操作符 (Enumeration Comparison Operators)

每一个枚举类型都固定提供下列预定义比较操作符:

```

bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);

```

对 `x op y` 求值的结果与对 `((U) X) op ((U) Y)` 求值的结果完全相同, 这里, `X` 和 `Y` 都是枚举类型 `E` 的表达式, 枚举类型 `E` 具有一个潜在类型 `U`, `op` 是这些比较操作符之一。也就是说, 枚举类型比较操作符只比较两个操作数的潜在整型值。

7.9.6 引用类型相等操作符 (Reference type Equality Operators)

预定义引用类型相等操作符是:

```

bool operator == (object x, object y);
bool operator != (object x, object y);

```

这些操作符返回比较这两个引用相等与否的结果。

由于预定义引用类型相等操作符接受类型 `object` 的操作数, 因此它们适用于所有不声明可用 `operator ==` 和 `operator !=` 成员的所有类型。而任何可用的用户自定义相等操作符都有效地隐藏预定义的引用类型相等操作符。

预定义的引用类型相等操作符要求操作数为 `reference-type` 或 `null` 值, 而且要求存在从一个操作数的类型到另一操作数类型的隐式转换。除非这两个条件都符合, 否则, 出现编辑期错误。这些规则中值得注意的是:

- 用预定义的引用类型相等操作符比较两个已知在编辑期不同的引用是错误的。例如, 如果操作数的编辑类型是两个类类型 `A` 和 `B`, 且 `A` 和 `B` 不互相派生 (不是从彼此派生而来的), 那么, 这两个操作数引用相同的对象是不可能的。因此, 这样的操作就是一个编辑期错误。
- 预定义的引用类型相等操作符不能比较值类型操作数。因此, 一个结构类型除非声明它自己的相等操作符, 否则, 不可能比较其值。
- 预定义的引用类型相等操作符的操作数绝不会发生封箱操作。执行这样的封箱操作是没有意义的, 对新分配封箱实例的引用一定与所有其它引用不同。

对于式子 `X==Y` 或 `X!=Y` 的任何操作来说, 如果存在任何可用的 `operator==` 或 `operator !=`, 那么, 操作符重载分解规则将选择那个操作符而不是预定义的引用类型相等操作符。然而, 通过把一或两个操作数显式转化为类型 `object` 来选择引用类型相等操作符是可能的。下面的例子

```

class Test
{
    static void Main() {
        string s = "Test";
    }
}

```



```

        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}

```

输出值为：

```

True
False
False
False

```

变量 S 和 T 指的是两个含有相同字母的不同的 string 实例。第一个比较输出 true，因为当两个操作数都是类型 string 时，选择预定义的字符串相等操作符（参见 7.9.7 节）。其余的比较都输出 false，因为当有一或两个操作数是类型 object 时，选择预定义的引用类型相等操作符。

注意上面的方法对于值类型来说无效。下面的例子：

```

class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        Console.WriteLine((object)i == (object)j);
    }
}

```

输出 false，因为这个计算引用被封装的 int 值的两个独立的实例。

7.9.7 字符串相等操作符（String Equality Operators）

预定义的字符串相等操作符是：

```

bool operator == (string x, string y);
bool operator != (string x, string y);

```

当下列条件之一符合时，就说这些 string 值相等：

- 两个值都是 null。
- 两个值都是对字符串实例的非 null 引用，这些字符串在其每一个字符位置都具有相似的长度及字符。

字符串相等操作符比较字符串的 values 而不是字符串的 references。当两个独立的字符串实例含有完全相同的字符顺序时，这两个字符串的值就是相等的，但其引用是不同的。如第 7.9.6 节所述，引用类型相等操作符可用于比较字符串引用而不能比较字符串的值。

7.9.8 委托相等操作符 (Delegate Equality Operators)

每一个委托类型 D 都固定提供下列预定义比较操作符：

```
bool operator ==(System.Delegate x, D y);
```

```
bool operator ==(D x, System.Delegate y);
```

```
bool operator !=(System.Delegate x, D y);
```

```
bool operator !=(D x, System.Delegate y);
```

两个委托实例相等的情况有以下几种：

- 如果两个委托实例有一个是 null，则只有两个都是 null 时，它们才相等。
- 如果两个委托实例有一个用另一个委托来说明，则只有当它们都用相同的委托实例来说明时，它们才能相等。否则，
- 如果两个委托实例有一个是非多 cast 委托，则只有当它们都是非多 cast 委托，且下条件有一个符合时，它们才能相等。两个都指的是相同的静态方法，或两个都指的是相同目标对象上相同的非静态方法。
- 如果两个委托实例有一个是多 cast 委托，那么，只有当它们的引用列表长度相同，且一个引用列表中的每一个委托都按顺序与另一个列表中的相应委托相等时，这两个委托实例才相等。

注意：根据上面的说明，不同类型的委托也可以被看作相等，只要它们具有相同的返回类型及参数类型。

7.9.9 is 操作符 (The is Operator)

is 操作符的作用是动态检查对象的运行期类型是否与给定类型相适应。操作 E is T 的结果是一个布尔型值，这个布尔型值表明 E 是否可通过一个引用转换、封箱转换或非封箱转换成功地转化为类型 T，这里，E 是一个表达式，T 是一个类型。此操作过程的求值情况如下：

- 如果 E 的编辑期类型与 T 相同，或者存在一个从 E 的编辑期类型到 T 的隐式引用转换（参见 6.1.4 节）或封箱转换（参见 6.1.5 节）；
 - 由于已知 E 是一给定类型，编辑器可对这种情况提出一个警告。
 - 如果 E 是一引用类型，那么，操作的结果就与 E != null 的求值结果相同。
 - 如果 E 是一值类型，则操作的结果就是 true。
- 否则，如果存在一个从 E 的编辑期类型到 T 的显式引用转换（参见 6.2.3 节）或非封箱转换（参见 6.2.4 节），则执行一个动态类型检查；
 - 如果 E 的值是 null，则结果为 false。

- 否则, 设 R 就是这个被 E 引用的实例的运行期类型。如果 R 和 T 类型相同, 而 R 又是一个引用类型且存在一个从 R 到 T 的显式转换, 或者如果 R 是一个值类型且 T 是一个被 R 执行的接口类型, 则其结果就是 true。
- 否则, 结果就是 false。
- 否则, 不可能有从 E 到类型 T 的引用或封箱转换。
 - 由于已知 E 绝不会是所给定类型, 因此, 编辑器可以此提出警告。
 - 操作过程的结果是 false。

注意: 操作符 is 只适用于引用转换、封箱转换和非封箱转换。其它的转换, 如用户自定义转换, is 操作符不适用。

7.9.10 as 操作符 (The as Operator)

as 操作符通过引用转换或封箱转换把一个值显式转化为一给定引用类型。与 cast 表达式 (参见 7.6.8 节) 不同, as 操作符绝不会出现异常。如果所表示的转换不可能发生, 则结果值就是 null。

在式子 E as T 的操作中, E 必须是一表达式且 T 必须是一引用类型。结果类型为 T, 且结果总是一个值。此操作的求值过程如下:

- 如果 E 的编辑期类型与 T 相同, 则结果就是 E 的值。
- 否则, 如果存在一个从 E 的编辑期类型到 T 的隐式引用转换或封箱转换, 则此转换就被执行且其结果就是这个操作的结果。
- 否则, 如果从 E 的编辑期类型到 t 存在一个显式引用转换, 则执行一个动态类型检查:
 - 如果 e 的值是 null, 那么, 结果就是这个具有编辑期类型 T 的 null;
 - 否则, 设 R 就是这个被 E 引用的实例的运行期类型。如果 R 和 T 的类型相同, 且 R 是一个引用类型, 又存在从 R 到 T 的隐式引用转换, 或者如果 R 是一个值类型, 而 T 是被 R 执行的一个接口类型, 那么, 结果就是由具有编辑期类型 T 所给定的引用。
 - 否则, 结果就是编辑期类型为 T 的 null 值。
- 否则, 所示的转换就不可能, 编辑时将出现错误。

注意: 操作符 as 只执行引用转换和封箱转换。其它的转换, 如用户自定义转换不能用 as 操作符, 应该用 cast 表达式来执行。

7.10 逻辑操作符 (Logical Operators)

操作符 &, ^ 及 | 称为逻辑操作符。

```
and-expression:
    equality-expression
    and-expression & equality-expression
```

```

exclusive-or-expression:
    and-expression
    exclusive-or-expression ^ and-expression

inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression | exclusive-or-expression

```

对于式子 `x op y` 的操作来说，重载分解的作用是选择一特定的操作符执行，这里 `op` 是这些逻辑操作符中的一个。操作数被转化为所选操作符的参数类型，且结果类型就是这个操作符的返回类型。

预定义的逻辑操作符将在以下几部分加以描述。

7.10.1 整数逻辑操作符 (Integer Logical Operators)

预先定义的整数逻辑操作符是：

```

int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);

```

操作符 `&` 计算两个操作数的位逻辑 `and`，操作符 `|` 计算两个操作数的位逻辑 `or`，操作符 `^` 计算两个操作数的位逻辑 `not` 包括 `or`。这些操作不会产生溢出。

7.10.2 枚举逻辑操作符 (Enumeration Logical Operators)

`E` 的每一个枚举类型都固定提供下列预定义逻辑操作符。

```

E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);

```

对 `x op y` 求值的结果与对 `(E)((U)(X) op ((U) Y))` 求值的结果相同，这里，`X` 和 `Y` 都是具有潜在类型 `U` 的枚举类型 `E` 的表达式，`op` 是这些逻辑操作符中的一个。也就是说，

枚举类型逻辑操作符只在两个操作数的潜在类型内执行逻辑操作。

7.10.3 布尔逻辑操作符 (Boolean Logical Operators)

预定义布尔逻辑操作符是:

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

如果 x 和 y 都是 `true`, 则 $x \& y$ 的结果为 `true`。否则, 其结果就是 `false`。如果 x 和 y 有一个是 `true`, 则 $x | y$ 的结果为 `true`。否则, 结果为 `false`。如果 x 是 `true`, y 是 `false` 或 x 是 `false`, y 是 `true`, 则 $x \wedge y$ 的结果为 `true`。否则, 结果为 `false`。当两个操作数的类型都是 `bool` 时, 操作符 \wedge 的计算结果与 `!` 相同。

7.11 附加条件逻辑操作符 (Conditional Logical Operators)

操作符 `&&` 和 `||` 称为附加条件逻辑操作符。有时也称为“短周期”逻辑操作符。

```
conditional-and-expression:
    inclusive-or-expression
conditional-and-expression && inclusive-or-expression

conditional-or-expression:
    conditional-and-expression
conditional-or-expression || conditional-and-expression
```

操作符 `&&` 和 `||` 是操作符 `&` 和 `|` 的附有条件的版本:

- 除非只有当 X 是 `true` 时, Y 才被求值, 否则, 操作 $X \& \& Y$ 与 $X \& Y$ 是一致的。

通过重载分解 (参见 7.2.4 节) 而执行的式子 $X \& \& Y$ 或 $X || Y$ 的操作就相当于把这个操作写成 $X \& Y$ 或 $X | Y$ 的形式。那么:

- 如果重载不能找到一个单个最佳操作符, 或重载分解选择这些预定义的整数逻辑操作符中的一个, 则错误发生。
- 否则, 如果所选的操作符是这些预定义的布尔逻辑操作符 (参见 7.10.2 节) 中的一个, 那么, 这个操作的执行过程如 7.11.1 节中所述。
- 否则, 所选的操作符就是一个用户自定义操作符, 此操作的执行过程见 7.11.2 节。

直接重载附有条件的逻辑操作符是不可能的。然而, 由于附加条件逻辑操作符是在固定的逻辑操作符条件下被求值的。因此, 在一定条件下, 固定的逻辑操作符的重载也被认为是附加条件逻辑操作符的重载。这将在 7.11.2 节进一步描述。

7.11.1 布尔条件逻辑操作符 (Boolean Conditional Logical Operators)

当 `&&` 或 `||` 的操作数是 `bool` 型, 或者当此操作数的类型不定义一可用的 `operator&` 或 `operator|`, 但定义一个到 `bool` 的隐式转换时, 此操作过程如下:

- $x \& \& y$ 的求值情况与 $x ? y : \text{false}$ 相同。也就是说，先对 x 进行求值，并转化为 bool 型。然后，如果 x 是 true ，对 y 求值并转化为 bool 型，且这就是操作的结果。否则，操作的结果就是 false 。
- 操作 $x // y$ 的求值情况与 $x ? \text{true} : y$ 相同。也就是说，先对 x 求值并转化为 bool 型。然后，如果 x 是 true ，操作的结果就是 true 。否则，对 y 求值并转化为 bool 型，且这就是操作的结果。

7.11.2 自定义条件逻辑操作符 (User-Defined Conditional Logical Operators)

当 $\&\&$ 或 $//$ 的操作数都是声明可用的用户自定义 $\text{operator}\&\&$ 或 $//$ 的类型时，下列两个条件必须符合，其中， t 就是声明所选操作符的类型：

- 返回类型及所选择操作符的每一个参数的类型都必须是 t 。也就是说，这个操作符必须计算类型 t 的两个操作数的逻辑 and 或逻辑 or ，且必须返回类型 t 的一个结果。
- t 必须包含 operator true 及 operator false 的声明。

如果这两个要求有一个不满足时，就会出现编辑期错误。否则， $\&\&$ 或 $//$ 操作过程的求值是通过把用户自定义 operator true 或 false 与所选的用户自定义操作符合并而进行的：

- 操作 $x \&\& y$ 的求值与 $t.\text{false}(x) ? t.\&(x, y)$ 相同，这里， $t.\text{false}(x)$ 是在 t 中声明的 operator false 的一个引用，而 $t.\&(x, y)$ 则是所选 $\text{operator}\&$ 的一个引用。也就是说，首先对 x 求值，并把 operator false 引用到结果中以确定 x 是否真的为假。然后，如果 x 确实为假，则操作的结果就是前面所计算的 x 的值。否则，对 y 求值，并把 $\text{operator}\&$ 引用到前面所计算的 x 和 y 值中以得到操作的结果。
- 对操作 $x // y$ 的求值与 $t.\text{true}(x) ? x : t./(x, y)$ 相同，这里， $t.\text{true}(x)$ 是在 t 中声明的 operator true 的一个引用，而 $t./(x, y)$ 是所选择的 $\text{operator}/$ 的一个引用。也就是说，先对 x 求值，并把 operator true 引用到结果中以确定 x 是否一定为真。然后，如果 x 确定为真，操作的结果就是前面所计算的 x 值。否则，对 y 求值，并把所选的 $\text{operator}/$ 引用到前面所计算的 x 和 y 中以得出操作的结果。

在这两个操作的任何一个中，由 x 给定的表达式只能被求值一次，而由 y 给定的表达式或不被求值或只能求值一次。

关于执行 operator true 和 operator false 类型的例子见 11.4.2。

7.12 条件操作符 (Conditional Operator)

操作符 $?$ ：称为条件操作符。有时也称为三元操作符。

```
conditional-expression:
    conditional-or-expression
    conditional-or-expression ? expression : expression
```

在式子 $B ? X : Y$ 的条件表达式中，首先对条件 B 求值。然后，如果 B 是 TRUE ，对 X 求值且它就是操作的结果。否则，对 Y 求值且其值就是操作的结果。一个条件表达式绝不会既对 X 求值，又对 Y 求值。

条件操作符是右结合的，意思是操作是从右到左进行的。例如，式子 $A ? B : C ? D : E$

的求值情况与 $A ? B : (操作 C ? D : E)$ 相同。

操作符 $?$ ：的第一个数必须是可转化为 `bool` 型的一个类型的表达式或执行 `operator true` 的一个类型的表达式。如果这两个要求都不能被满足，则出现编辑期错误。

操作符 $?$ ：的第二和第三个操作数控制条件表达式的类型。设 X 和 Y 分别为第二和第三个操作数的类型，则：

- 如果 X 和 Y 类型相同，那么，其类型就是条件表达式的类型。
- 否则，如果存在一个从 X 到 Y 的隐式转换（参见 6.1 节），但从 Y 到 X 不存在隐式转换，那么， Y 就是这个条件表达式的类型。
- 否则，如果存在一个从 Y 到 X 的隐式转换（参见 6.1 节）而从 X 到 Y 不存在隐式转换，则 X 就是这个条件表达式的类型。
- 否则，表达式的类型不能确定，编辑时将出现错误。

式子 $B ? X : Y$ 的一个条件表达式的运行期处理过程包括以下几步：

- 首先，对 B 求值，并确定 B 的 `bool` 型值：
如果存在一个从 b 的类型到 `bool` 型的隐式转换，那么，执行这个隐式转换以得到一个 `bool` 值。否则，引用由 b 的类型定义的 `operator true` 以得到一个 `bool` 值。
- 如果由上面的步骤所产生的 `bool` 值是 `true`，那么，对 x 求值并把 x 转化为这个条件表达式的类型，且这就是此条件表达式的结果。
- 否则，对 Y 求值并把 Y 转化为这个条件表达式的类型，且这就是此条件表达式的结果。

7.13 赋值操作符 (Assignment Operators)

赋值操作符给变量、属性或索引成员赋一个新值。

```
assignment:
    unary-expression assignment-operator expression

assignment-operator: one of
    = += -= *= /= %>= &= != ^= <<= >>=
```

一个赋值的左操作数必须是作为变量或属性访问、索引访问的表达式。

操作符 $=$ 称为 `simple assignment operator`。它把右操作数的值赋给由左操作数所给定的变量、属性或索引成员。关于简单赋值操作符将在 7.13.1 节中描述。

通过在一个二进制操作符的前面加上一个 $=$ 字符而形成的操作符称为 `compound assignment operators`。此操作符对两个操作数进行操作，并把所得值赋给由左操作数所给定的变量、属性或索引成员。复合赋值操作符将在 7.13.2 节中描述。

赋值操作符是右结合的，意思是说，操作是从右到左进行的。例如，对式子 $a=b=c$ 的求值与 $a=(b=c)$ 相同。

7.13.1 简单赋值 (Simple Assignment)

操作符 $=$ 称为简单赋值操作符。在简单赋值过程中，右操作数必须是其类型可被隐式转

换为右操作数类型的一个表达式。此操作把右操作数的值赋给由左操作数所给定的变量、属性或索引元素。

简单赋值表达式的结果就是赋给左操作数的值。此结果与左操作数的类型相同，并且总被看作一个值。

如果左操作数是一个属性或索引访问，那么，这个属性或索引必须具有一个 SET 访问函数。否则，编辑时将出现错误。

式子 $X=Y$ 的一个简单赋值的运行期处理包括以下几步：

- 如果 X 是一个变量：
 - 对 X 求值产生这个变量。
 - 对 Y 求值，且如果需要的话，通过隐式转换把它转化为 X 的类型。
 - 如果由 X 给定的变量是一个 reference-type 的数组元素，则执行一个运行期检查以确保 y 的计算值与含有元素 x 的数组实例相适应。如果 y 是 null，或者存在一个从被 y 所引用的实例的实际类型到包含 x 的数组实例的实际元素类型的隐式引用转换，则检查成功。否则，显示 arraytypemismatch exception。
 - 把由 y 的求值和转换过程所得到的值存储在由 x 的求值过程所给定的空间内。
- 如果 x 是一个属性或索引访问：
 - 对与 x 有关的实例表达式（如果 x 不是 static）和自变量列表（如果 x 是一个索引访问）进行求值，并把这些值用在后面的 set 访问函数引用过程中。
 - 对 y 求值，如果必要通过隐式转换把它转化为 x 的类型。
 - 引用 x 的 set 访问函数， y 的计算值作为其 value 自变量。

数组方差规则（参见 12.5 节）允许数组类型 $a[]$ 的值是对数组类型 $b[]$ 的一个实例的引用，假设从 b 到 a 的值是对数组类型 $b[]$ 的一个实例的引用，且从 b 到 a 存在一个隐式引用转换。由于这些规则，对 reference-type 的一个数组元素的赋值需要进行运行期检查以确保被赋的值适合于这个数组实例。在下面的例子中：

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;           // Ok
oa[1] = "Hello";        // Ok
oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

最后的赋值显示 array type mismatch exception，因为 arraylist 的实例不能存储在 string[] 的元素中。

当在 struct-type 中声明的属性或索引是赋值的目标时，与这个属性或索引访问有关的实例表达式必须是一个变量。如果这个实例变量是一个值，则出现编辑期错误。

给定一个声明：

```
struct Point
{
    int x, y;
```



```
public Point(int x, int y) {
    this.x = x;
    this.y = y;
}

public int X {
    get { return x; }
    set { x = value; }
}

public int Y {
    get { return y; }
    set { y = value; }
}
}

struct Rectangle
{
    Point a, b;
    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }
    public Point A {
        get { return a; }
        set { a = value; }
    }
    public Point B {
        get { return b; }
        set { b = value; }
    }
}
}
```

在下面的例子中：

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

允许对 P.X, P.Y, R.A 及 R.B 的赋值，因为 P 和 R 都是变量。然而，在下面的例子中：

```

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;

```

所有的赋值都是无效的，因为 R.A 和 R.B 不是变量。

7.13.2 复合赋值 (Compound Assignment)

运用二进制操作符重载分解的式子 $xop=y$ 的执行过程就相当于把它书写为 $xopy$ 之后的操作过程。那么：

- 如果所选操作符的返回类型可被 *implicitly* 转化为 x 的类型，则这个操作的求值同 $x=xopy$ ，除非 x 只能被赋值一次。
- 否则，如果所选操作符是一个预定义的操作符，且其返回类型可被 *explicitly* 转化为 x 的类型，而 y 可被 *implicitly* 转化为 x 的类型，则这个操作的求值方式就是 $x=(t)(xipy)$ ，其中， t 是 x 的类型，除非 x 只能被赋值一次。
- 否则，这个复合赋就是无效的，且出现编辑期错误。

术语“只能被赋值一次”的意思是说，在 $xopy$ 的求值过程中，由 x 组成的任何结果都被临时保留，并在对 x 赋值时重新被利用。例如，在赋值 $a() [b()] += c()$ 中， a 是返回 $int[]$ 的一种方法， b 和 c 都是返回 int 的方法，这些方法只按顺序 a 、 b 、 c 被引用一次。

当一复合赋值的左操作数是属性访问或索引访问时，这个属性或索引必须既有 *get* 访问函数，又有 *set* 访问函数。否则，出现编辑期错误。上面的第二条规则允许在特定的上下文中， $xop=y$ 以 $x=(t)(xopy)$ 的形式被求值。这条规则使得预先定义的操作符可被作为复合操作符来使用，条件是左操作数的类型是 *sbyte*、*byte*、*short*、*ushort* 或 *char*。甚至当两个自变量都是那些类型中的一种时，预定义的操作符也产生类型为 *int* 的一个结果，如 7.2.6.2 节所述。因此，如果没有 *cast*，不可能把结果赋给左操作数。

这条预定义操作符规则的直观效果只是：如果操作 $xopy$ 和 $x=y$ 被允许，那么 $xopy=y$ 也被允许。在下面的例子中：

```

byte b = 0;
char ch = '\0';
int i = 0;

b += 1;           // Ok
b += 1000;        // Error, b = 1000 not permitted
b += i;           // Error, b = i not permitted
b += (byte)i;     // Ok

ch += 1;          // Error, ch = 1 not permitted
ch += (char)1;    // Ok

```

每一个错误的直接原因就是相应的简单赋值是错误的。

7.14 表达式 (Expression)

表达式(expression)或是一个 conditional-expression, 或者是一个 assignment。

```
expression:
    conditional-expression
    assignment
```

7.15 常量表达式 (Constant Expressions)

一个常量表达式 (constant-expression) 就是一个在编辑期可被完全求值的表达式。

```
constant-expression:
    expression
```

常量表达式的类型有以下几种: sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string 任何枚举类型或 null 型。

在常量表达式中允许出现下列组分:

- 字母 (包括 null 字母)。
- 对类或结构类型的 const 元素的引用。
- 对枚举类型成员的引用。
- 带括弧的子表达式。
- cast 表达式, 假设目标类型是上面所列类型中的一种。
- 预定义的一元操作符+, - ! 及~。
- 预定义的二元操作符+, -, *, /, %, 《, 》, &, /, ^, &&, //, ==, !=, <, >, <= 及>=, 假设每一个操作数都是上面所列的类型。
- 条件操作符?:。

如果一个表达式是上面所列类型中的一种且只包含上面所列组分, 那么, 这个表达式就在运行期被求值。即使这个表达式是一个较大的包括非常量组分的表达式的子表达式, 也存在这种情况。

常量表达式的编辑期求值所用的规则与非常量表达式的运行期求值所用的规则相同, 除非运行期求值出现异常, 那么, 编辑期求值也会出现编辑错误。

除非一个常量表达式在 unchecked 上下文中被显式替换, 否则, 发生在整类型算术操作中的溢出及表达式的编辑期求值过程中的转换总是导致编辑期错误 (参见 7.5.12 节)。

- 常量声明 (参见 10.3 节)。
- 枚举成员声明。
- switch 语句中的 case 符号 (参见 8.7.2 节)。
- goto case 语句 (参见 8.9.3 节)。
- 包括一个初始化函数的数组产生表达式 (参见 7.5.10.2) 中的维长度。
- 属性 (参见 17 章)。

隐式常量表达式转换（参见 6.1.6 节）允许把类型 `int` 的常量表达式转化为 `sbyte`、`byte`、`short`、`ushort`、`uint` 或 `ulong`，假设这个常量表达式的值在目的类型的取值范围内。

7.16 布尔表达式（Boolean Expressions）

一个布尔表达式（`boolean-expression`）是所得结果类型为 `bool` 的一个表达式。

`boolean-expression`;

`expression`

一个 `if-statement`（参见 8.7.1 节）、`while-statement`（参见 8.8.1 节）、`do-statement`（参见 8.8.2 节）或 `for-statement`（参见 8.8.3 节）的控制条件表达式就是一个 `boolean-expression`。操作符 `?:` 的控制条件表达式与 `boolean-expression` 遵守相同的规则，但由于操作符执行方面的原因而被归类为一个 `conditional-or-expression`。

一个 `boolean-expression` 要求是这样一个类型，即它可被隐式转化为 `bool` 或执行 `operator true`。如果这两个要求都不能满足，则将出现编辑期错误。

当一个布尔表达式是一个不能被隐式转化为 `bool`，但确实能执行 `operator true` 的类型时，在这个表达式被求值之后，就要引用由此类型提供的 `operator true` 执行以得到一个 `bool` 值。11.4.2 节中的 `dbbool` 结构类型就是执行 `operator true` 的一个类型的例子。

第8章 语 句

C#提供了很多语句。这些语句绝大多数对于曾在 C 和 C++中编程的开发者来说都是熟悉的。

```
statement:
    labeled-statement
    declaration-statement
    embedded-statement

embedded-statement:
    block
    empty-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    try-statement
    checked-statement
    unchecked-statement
    lock-statement
    using-statement
```

无终止的 `embedded-statement` 被用在那些包含其它语句之内的语句中。在这些上下文中，`embedded-statement` 而不是 `statement` 的运用不包括声明语句和标号语句的运用。例如，代码

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

是错误的，因为一个 `if` 语句的分支要求一个 `embedded-statement` 而不是 `statement`。如果这种代码被允许，则变量 `i` 将被声明，但它绝不能被应用。

8.1 结束点和可到达性 (End Points And Reachability)

每个语句都有一个 `end point`。一个语句的结束点的字面意思就是紧跟在这个语句之后的那个位置。当控制到达一嵌套语句的终点时，复合语句（包括嵌套语句的语句）的执行规则规定了该怎么做。例如，在一个块中，当控制到达一个语句的终点时，它就被转到这个块中的另一个语句。

如果一个语句通过执行可到达，那么就说明这个语句是 `reachable`。相反，如果一个语句不可能被执行，那么就说明这个语句是 `unreachable`。在下面的例子中第二个 `console.WriteLine` 引不可能到达，因为这个语句不可能被执行：

```
void F() {  
    Console.WriteLine("reachable");  
    goto Label;  
    Console.WriteLine("unreachable");  
    Label:  
    Console.WriteLine("reachable");  
}
```

如果编辑器确定一个语句不可到达，那么，就有一个警告显示。但语句不能到达不是一个错误。

为了确定某一特定语句或结束点是否能可到达，编辑根据每一条语句的可到达规则进行了流动分析。流动分析考虑了控制语句行为的常量表达式（参见 7.15 节）的值，但不考虑非常量表达式的可能值。也就是说，为了控制流动分析，认为一给定类型的非常量表达式可具有该类型的任何可能值。在下面的例子中：

```
void F() {  
    const int i = 1;  
    if (i == 2) Console.WriteLine("unreachable");  
}
```

if 语句的布尔表达式是一个常量，因为操作符 == 的两个操作数都是常量。常量表达式在编辑被求值，产生值 false，因此，对 Console.WriteLine 的引用被认为是不可到达的。然而，如果 i 变成一个局布变量

```
void F() {  
    int i = 1;  
    if (i == 2) Console.WriteLine("reachable");  
}
```

则对 Console.WriteLine 的引用就被认为是可到达的，即使它实际上不可能被执行。

一个函数成员的 block 总被认为是可到达的。通过准确地把握一个块中每条语句的可到达性规则，可以确定任何给定语句的可到达性。

在下面的例子中：

```
void F(int x) {  
    Console.WriteLine("start");  
    if (x < 0) Console.WriteLine("negative");  
}
```

第一个 Console.WriteLine 的可到达性确定如下：

- 首先，由于方法 f 的块是可到达的，因此，第一个 Console.WriteLine 语句也是可到达的。
- 其次，因为第一个 Console.WriteLine 语句是可到达的，其结束点也是可到达的。
- 再次，因为第一个 Console.WriteLine 语句的结束点是可到达的，if 语句也是可到达的。

- 最后, 因为 if 语句的布尔表达式不含有常量值 false, 所以, 第二个 Console.WriteLine 语句是可到达的。

在两种情况下, 语句的结束点可到达是错误的:

- 因为 switch 语句不允许一个转换部分通过下一个转换部分, 所以, 一个转换部分语句列表的结束点可到达是错误的, 如果这种错误发生, 就明显表明忽略了一个 break 语句。
- 求值函数成员块的结束点不可到达。如果这种错误发生, 很明显是因为忽略了一个 return 语句。

8.2 块 (Blocks)

一个 Block 允许在一个语句出现的上下文中书写多个语句。

```
block:
{statement-list}
```

一个 Block 是由大括号内一个可选择的 statement-list 组成的。如果这个语句列表被省略, 那么, 这个块就是空的。

块可以包括声明语句 (参见 8.5 节)。在一块中声明的局部变量或常量的范围从这个声明延伸到块的结束。

块内表达式上下文中所用的名字的意思必须总是相同的 (参见 7.5.2.1 节)。块的执行如下:

- 如果块是空的, 则控制就被转到块的结束。
- 如果块不是空的, 则控制就被转到语句列表。如果控制到达语句列表的结束点, 则控制被转到块的结束点。
- 如果块本身是可到达的, 则其语句列表也是可到达的。
- 如果一个块是空的或者语句列表的结束点是可到达的, 则这个块的终点就是可到达的。

一个语句列表 (statement list) 包括一个或多个按顺序书写的语句。语句列表发生在 Blocks (参见 8.2 节) 及 switch-blocks (参见 8.7.2 节) 中。

```
statement-list:
statement
statement-list statement
```

一个语句列表通过把控制转到第一条语句而被执行。当控制到达一条语句的终点时, 控制就被转到下一条语句。当控制到达最后一条语句的结束点时, 控制就被转到该语句列表的结束点。

如果至少下列条件之一符合, 那么, 在一个语句列表中的某条语句就是可到达的:

- 该语句是第一条语句且这条语句本身是可到达的。
- 先前语句的结束点是可到达的。

- 这条语句是一标号语句，且这个标号被一可达到的 goto 语句引用。

如果列表中最后一条语句的结束点是可达到的，那么，一个语句列表的终点就是可达到的。

8.3 空语句 (The Empty Statement)

一个空语句 (empty-statement) 不执行任何操作。

empty-statement

当在一上下文中，需要一条语句而又没有操作可执行时，就用一条空语句。

空语句的执行只是把控制转到这条语句的结束点。因此，如果一条空语句是可达到的，那么，这条空语句的结束点就是可达到的。

当书写一个带有 null 主体的 while 语句时，可用一个空语句。

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())

}
```

空语句也可用来声明一个在块结束符 “}” 之前的符号。

```
void F() {
    ...

    if (done) goto exit;
    ...

    exit: ;
}
```

8.4 标号语句 (Labeled Statements)

标号语句 (labeled-statement) 允许一条语句用一标号作为前缀。标号语句可以在块内出现，但不能作为嵌套语句。

labeled-statement

identifier: statement

标号语句声明一个具有由 identifier 给定的名字的标号。标号的范围就是声明该标号的块，包括任何嵌套块。具有相同名字的两个标号的范围互相重迭是错误的。

一个标号可从其范围内的 goto 语句 (参见 8.9.3 节) 开始被引用。也就是说，goto 语句可以在块内及块外转换控制，但绝不能转入块。

标号具有它们自己的声明空间且不干扰其它的标识符。例如


```

int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}

```

是有效的，名字 X 既是一个参数，又是一个标号。

一标号语句的执行与其标号后面语句的执行完全一致。

除由控制的正常流动提供的可达性外，如果一个标号被一可达到的 goto 语句引用，那么，这个标号语句就是可达到的。

8.5 声明语句 (Declaration Statements)

一个声明语句 (declaration-statement) 声明一个局部变量或常量。声明语句可以出现在块中，但不能作为嵌套语句。

```

declaration-statement:
    local-variable-declaration ;
    local-constant-declaration ;

```

8.5.1 局部变量声明 (Local Variable Declarations)

一个局部变量声明 (local-variable-declaration) 声明一或多个局部变量。

```

local-variable-declaration:
    type variable-declarators

variable-declarators:
    variable-declarator
    variable-declarators , variable-declarator

variable-declarator:
    identifier
    identifier = variable-initializer

variable-initializer:
    expression
    array-initializer

```

局部变量声明的类型就限定了由这个声明引入的变量的类型。类型的后面是一个 variable-declarator 的列表，其中每一个 variable-declarator 引入一个新的变量。一个 variable-declarator 包括一个作为该变量名的 identifier，其后是一个可选的符号“=”及一赋给这个变量初始值的 variable-initializer。

在表达式中，局部变量的值可用 simple-name (参见 7.5.2 节) 来得到，且用 assignment (参见 7.13 节) 来修改。局部变量必须在取得其值的每一个空间被明确赋值 (参见 5.3 节)。

局部变量的范围开始于声明中它的标识符之后，延伸至包含这个声明的块的终点。在局部变量范围内，用同一个名字声明另一个局部变量或常量是错误的。

声明多个变量的局部变量声明等同于具有相同类型的单个变量的多次声明。而且，在局部变量声明过程中的变量初始化与直接插入到该声明后面的赋值语句完全一致。

例如：

```
void F() {
    int x = 1, y, z = x * 2;
}
```

与

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

是完全一致的

8.5.2 局部常量声明 (Local Constant Declarations)

一个局部常量声明 (Local-constant-declaration) 声明一个或多个局部常量。

```
local-constant-declaration:
const type constant-declarators
constant-declarators:
constant-declarator
constant-declarators , constant-declarator
constant-declarator:
identifier=constant-expression
```

局部常量声明的类型指定了由这个声明所引入的常量的类型。该类型后面是一个 `constant-declarators` 列表，其中，每一个 `constant-declarator` 引入一个新的常量。`constant-declarator` 包括一个作为该常量名字的 `identifier`，其后是符号“=”，后面跟一个给出这个常量值的 `constant-expression` (参见 7.15 节)。

局部常量声明的 `type` 及 `constant-expression` 必须与同常量成员声明 (参见 10.3 节) 遵守相同的规则。

在表达式中，局部常量的值可用 `simple-name` (参见 7.5.2 节) 来得到。

局部常量的范围是从它的声明到包含这个声明的块的结束。它不包括提供其值的 `constant-expression`。在局部常量的范围内，用同一个名字声明另一个局部变量或常量是错误的。

8.6 表达式语句 (Expression Statements)

一个表达式语句 (expression-statement) 对给定的表达式进行求值。由这个表达式所计算的值 (如果有) 被删除。

```
expression-statement:
    statement-expression ;

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression
```

并不是所有的表达式都可以作为语句。尤其是像 $X+Y$ 及 $X=1$ 这样没有副效应, 只计算一个值 (这个值将被删去) 的表达式不能作为语句。

Issue

Define "side effect".

表达式语句的执行对所含的表达式求值, 并把控制转到这个表达式语句的结束点。

8.7 选择语句 (Selection Statements)

选择语句为基于控制表达式值之上的执行选择一条语句。

```
selection-statement:
    if-statement
    switch-statement
```

8.7.1 if 语句 (The If Statement)

if 语句为基于一个布尔表达式值之上的执行选择一语句。

```
if-statement:
    if ( boolean-expression ) embedded-statement
    if ( boolean-expression ) embedded-statement else
    embedded-statement

boolean-expression:
    expression
```

else 部分与前面最近的 if 语句有关, 这条 if 语句还不具有 else 部分。因此, 式子 IF (x) IF (Y) F (); else G (); 的一个 if 语句与下面的相同:

```

if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}

```

if 语句的执行如下:

- 对 `boolean-expression` (参见 7.16 节) 求值。
- 如果这个布尔表达式的值是 `true`, 则控制被转到第一个嵌套语句。当控制到达那个语句的结束点时, 控制就被转到该 if 语句的结束点。
- 如果布尔表达式的值是 `false` 且存在 `else` 部分, 则控制被转到第二个嵌套语句。当控制到达那个语句的终点时, 就被转到 if 语句的结束点。
- 如果布尔表达式的值是 `false` 且不存在 `else` 部分, 则控制被转到 if 语句的结束点。

如果一个 if 语句是可到达的且布尔表达式不具有常量值 `false`, 那么, 这个 if 语句的第一个嵌套语句就是可到达的。

如果一个 if 语句是可到达的, 且布尔表达式不具有常量值 `true`, 那么, 这个 if 语句的第二个嵌套语句就是可到达的。

如果至少一个嵌套语句的结束点是可到达的, 则这个 if 语句的结束点就是可到达的。另外, 如果一个 if 语句没有 `else` 部分且是可到达的, 且布尔表达式不具有常量值 `true`, 则这个 if 语句的结束点就是可到达的。

8.7.2 switch 语句 (The Switch Statement)

switch 语句执行与控制表达式的值有关的语句。

```

switch-statement:
    switch ( expression ) switch-block

switch-block:
    { switch-sections_opt }

switch-sections:
    switch-section
    switch-sections switch-section

switch-section:
    switch-labels statement-list

switch-labels:
    switch-label
    switch-labels switch-label

```

```
switch-label:
    case constant-expression :
    default :
```

一个 switch-statement 包括关键词 switch，其后是括弧内的表达式（称为 switch 表达式），然后是一个 switch-block。这个 switch-block 包括大括号内的零或多个 switch-sections。每一个 switch-section 包括一或多个后面带有一个 statement-list 的 switch-labels。

一个 switch 语句的 governing type 是由这个 switch 表达式确定的。如果这个 switch 表达式的类型是 sbyte、byte、short、ushort、int、uint、long、ulong、char、string 或 enum-type，那么，它就是这个 switch 语句的控制类型。否则，必须只存在一个用户自定义隐式转换（参见 6.4 节），这个转换才可把这个 switch 表达式的类型转化为下列可能的支配类型之一：sbyte、byte、short、ushort、int、uint、long、ulong、char、string。如果这样的隐式转换不存在或多于一个，则出现编辑错误。

每一个 case 标号的常量表达式都必须表示可被隐式转化为 switch 语句支配类型的一个值。在同一个 switch 语句中，如果两个或更多的 case 标号规定相同的常量值，则出现编辑错误。在一个 switch 语句中，至多只能有一个 default 标号。

switch 语句的执行过程如下：

- 对 switch 表达式求值并把它转化为支配类型。
- 如果在一个 case 标号中规定的常量有一个与这个 switch 表达式的值相等，那么，控制就被转到相对的 case 标号后面的语句列表。
- 如果没有与 switch 表达式的值相匹配的常量且存在 default 标号，那么，控制就被转到这个 default 标号后面的语句列表。
- 如果没有与这个 switch 表达式的值相匹配的常量且不存在一个 default 标号，则控制就被转到这个 switch 语句的结束点。

如是 switch 部分语句列表的结束点是可达的，则出现编辑错误。这就是“不通过”规则。下面的例子是有效的，因为转换部分都没有可达的结束点：

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}
```

与 C 和 C++ 不同，switch 部分的执行不允许“通过”下一个 switch 部分，且这个例子是错误的：

```
switch (i) {  
    case 0:  
        CaseZero();  
    case 1:  
        CaseZeroOrOne();  
    default:  
        CaseAny();  
}
```

当一个 `switch` 部分的执行后面紧跟着是另一个 `switch` 部分的执行时，必须用一个显式 `goto case` 或 `goto default` 语句；

```
switch (i) {  
    case 0:  
        CaseZero();  
        goto case 1;  
    case 1:  
        CaseZeroOrOne();  
        goto default;  
    default:  
        CaseAny();  
        break;  
}
```

在一个 `switch-section` 中允许有多个标号。下面的例子是合法的：

```
switch (i) {  
    case 0:  
        CaseZero();  
        break;  
    case 1:  
        CaseOne();  
        break;  
    case 2:  
    default:  
        CaseTwo();  
        break;  
}
```

这个例子不违反“不通过”规则，因为标号 `case` 和 `default` 都是同一个 `switch-section` 的部分。

“不通过”规则禁止当 `break` 语句被忽略时发生在 C 及 C++ 中窃听的普通类。由于这条规则，一个 `switch` 语句的转换部分也可以被任意重排而不影响这个语句的行为。例如，上面

的 `switch` 语句的部分可被颠倒而不影响这个语句的行为。

```
switch (i) {  
    default:  
        CaseAny();  
        break;  
    case 1:  
        CaseZeroOrOne();  
        goto default;  
    case 0:  
        CaseZero();  
        goto case 1;  
}
```

转换部分的语句列表典型的结束语句是 `break`、`goto`、`case` 或 `goto`、`default`，但是允许此语句列表的任何一个没有结束点的组成部分不可到达。例如，已知由布尔表达式 `true` 控制的 `while` 语句绝不会到达其结束点。同样，`throw` 或 `return` 语句总是把控制转移到其它地方而绝不会到达其结束点。因此，下面的例子是有效的：

```
switch (i) {  
    case 0:  
        while (true) F();  
    case 1:  
        throw new ArgumentException();  
    case 2:  
        return;  
}
```

`switch` 语句的支配类型可以是 `string`。例如：

```
void DoCommand(string command) {  
    switch (command.ToLower()) {  
        case "run":  
            DoRun();  
            break;  
        case "save":  
            DoSave();  
            break;  
        case "quit":  
            DoQuit();  
            break;  
        default:  
            InvalidCommand(command);  
            break;  
    }
```

```

    }
}

```

同字符串相等操作符（参见 7.9.7 节）一样，switch 语句是 case 敏感型的且只有当 switch 表达式字符串与一个 case 标号常量完全匹配时，这样的 switch 语句才能执行一给定的 switch 部分。如上例所示，一个 switch 语句可以通过把其表达式字符串转化为小写的 case 及把所有的 case 标号常量都以小写 case 的形式来书写而变成 case 敏感型。

当一个 switch 语句的支配类型是 string 时，允许值 null 作为一个 case 标号常量。一个 switch-block 可以包括声明语句。在 switch 块中声明的局部变量或常量的范围从声明开始到该 switch 块的结束。在一个 switch 块内，用在同一个表达式上下文中的某一个名字的意思必须总是相同的。如果一个 switch 语句是可到达的，且至少符合下列条件之一，则这个给定的 switch 部分的语句列表就是可达到的：

- 该 switch 表达式是一个非常量值。
- 该 switch 表达式是与其 switch 部分的一个 case 标号相匹配的常量值。
- 该 switch 表达式是一个不与任何 case 标号匹配的常量值，且此 switch 部分包括 default 标号。
- 该 switch 部分的 switch 标号被一可达到的 goto case 或 goto default 语句引用。

如果一条 switch 语句至少符合下列条件之一，则该 switch 语句的就是可达到的：

- 这条 switch 语句包括一个可达到的 break 语句，它使得这条 switch 语句得以出现。
- 这条 switch 语句是可达到的，这个 switch 表达式是一个非常量值，且不存在 default 标号。
- 这条 switch 语句是可达到的，switch 表达式是一个常量值且不与任何 case 标号相匹配，不存在 default 标号。

8.8 iteration 语句（Iteration Statements）

iteration 语句重复执行一个嵌套语句。

```

iteration-statement:
while-statement
do-statement
for-statement
foreach-statement

```

8.8.1 while 语句（The While Statement）

while 语句有条件地把一个嵌套语句执行零或多次。

```

while-statement:
while (boolean-expression) embedded-statement

```


while 语句的执行过程如下:

- 对 boolean-expression (参见 7.16 节) 求值。
- 如果这个布尔表达式产生一个 true 值, 则控制被转到这个嵌套语句。当控制到达这条嵌套语句的结束点时(可能从一个 continue 语句的执行开始), 控制就被转到 while 语句的开始。
- 如果布尔表达式产生一个 false 值, 则控制被转到这条 while 语句的结束点。

在一 while 语句的嵌套语句内, break 语句(参见 8.9.1 节)的作用是把控制转到这条 while 语句的结束点(这样, 就结束了这个嵌套语句的重复), 而 continue 语句(参见 8.9.2 节)的作用是把控制转到这个嵌套语句的结束点(这样就执行此 while 语句的另一个重复)。

如果一条 while 语句是可到达的且布尔表达式不具有常量值 false, 则这条 while 语句的嵌套语句就是可达到的。如果一条 while 语句至少符合下列条件之一, 则这条 while 语句的结束点就是可达到的:

- 该 while 语句包括一个可达到的 break 语句, 它使得 while 语句得以存在。
- 该 while 语句是可达到的, 且这个布尔表达式不具有常量值 true。

8.8.2 do 语句 (The Do Statement)

do 语句有条件地把一个嵌套语句执行一次或多次。

```
do-statement:
    do embedded-statement while ( boolean-expression ) ;
```

do 语句的执行过程如下:

- 当控制到达这条嵌套语句的结束点时(可能是从执行一个 continue 语句开始), 对其 boolean-expression 求值。如果布尔表达式结果为 true, 则控制就被转到这条 do 语句的开始。否则, 控制就被转到其结束点。

在一个 do 语句的嵌套语句内, 可用 break 语句把控制转到这条 do 语句的结束点(这样就结束了此嵌套语句的重复), 而 continue 语句则可把控制转到该嵌套语句的结束点(这样就执行此 do 语句的另一个重复)。如果一条 do 语句是可到达的, 那么, 这条 do 语句的嵌套语句也是可达到的。如果至少符合下列条件之一, 那么, 一个 do 语句的结束点就是可达到的:

- 该 do 语句包含一个可达到的 break 语句, 它使得此 do 语句得以存在。
- 其嵌套语句的结束点是可达到的, 且其布尔表达式不具有常量值 true。

8.8.3 for 语句 (The For Statement)

for 语句对一系列初始化表达式求值, 且当条件符合时, 重复执行一条嵌套语句并对一系列重复表达式求值。

```
for-statement:
    for ( for-initializeropt , for-conditionopt
          for-iteratoropt ) embedded-statement
for-initializer:
```

```

local-variable-declaration
statement-expression-list
for-condition:
    boolean-expression
for-iterator:
    statement-expression-list
    statement-expression-list:
    statement-expression
    statement-expression-list , statement-expression

```

for-initializer(如果存在)包括由逗号隔开的 local-variable-declaration 或 statement-expressions (参见 8.6 节) 的一个列表。由 for-initializer 声明的局部变量的范围从这个变量的 variable-declarator 开始, 至这条嵌套语句的结束点。这个范围包括 for-condition 及 for-iterator。

for-condition (如果存在) 必须是一个 boolean-expression。

for-iterator (如果存在) 则是由被逗号隔开的一个 statement-expressions 列表组成的。

for 语句的执行过程如下:

- 如存在 for-initializer, 则其变量初始化及语句表达式按其书写顺序被执行。此步只执行一次。
- 如果存在一个 for-condition, 则对它求值。
- 如果 for-condition 不存在或其求值结果为 true, 则控制被转到这条嵌套语句。当控制到达这条嵌套语句的结束点时(可能是从执行一个 continue 语句开始), for-iterator 的表达式(如果有)按顺序被求值, 且另一个重复被执行, 开始于上面步骤中对 for-condition 的求值。
- 如果 for-condition 存在, 且求值结果为 false, 则控制就被转到这条 for 语句的结束点。

在一条 for 语句的嵌套语句内, break 语句的作用是把控制转到这条 for 语句的结束点(这样, 就结束于此嵌套语句的重复), 而 continue 语句的作用则是把控制转到这个嵌套语句的结束点(这样便执行此 for 语句的另一个重复)。

如果下列条件之一符合, 就说一条 for 语句的嵌套语句是可到达的:

- 该 for 语句是可到达的, 且不存在 for 条件 (for-condition)。
- 该 for 语句是可到达的, 存在 for-condition 且未出现常量值 false。

如果下列条件之一符合, 那么就说一条 for 语句的结束点可到达:

- 该 for 语句包括一个可到达的 break 语句, 它使得该 for 语句得以存在。
- 该 for 语句是可到达的且存在一个 for-condition, 未出现常量值 true。

8.8.4 for each 语句 (The For Each Statement)

for each 语句枚举一个集合的元素, 并为此集合的每一个元素执行一个嵌套语句。

```
foreach-statement:
    foreach ( type identifier in expression )
        embedded-statement
```

for each 语句的 type 和 identifier 声明这个语句的 iteration variable。这个循环变量与只读局部变量一致，其范围延伸至这个嵌套语句内。在一条 foreach 语句执行期过程中，该循环变量表示当前正在执行的一个循环的全体成员，如果该嵌套语句试图对此循环变量进行赋值或把它当作一个 ref 或 out 参数来传递，那么，就会出现编辑错误。

foreach 语句的 expression 类型必须是一个集合类型（如下面所定义），且必须存在一个从该集合的元素类型到此循环变量类型的显式转换（参见 6.2 节）。

如果下列所有条件都符合，那么，就说类型 C 是一个 collection type:

- C 包括一个具有 get enumerator () 签名的 public 实例方法，这个方法返回一个 struct-type、class-type 或 interface-type，以下称为 E。
- E 包括一个具有 move next () 签名且返回类型为 bool 的 public 实例方法。
- E 包括一个命名为 current 的 public 且允许阅读实例属性。该属性的类型就是这个集合类型的 element type。

类型 system array(参见 12.1.1 节)是一个集合类型，且由于所有的数组类型都是从 system.array 得到的，因此，任何数组类型表达式在 foreach 语句以升序列举数组元素，从 0 开始，到 length-1 结束。对于多维数组而言，首先以最右边的维的升序索引。

一条 foreach 语句的执行过程如下：

- 对集合表达式求值得到该集合类型的一个实例。在下面的表述中，这个实例用 C 来表示。如果 C 是一个 reference-type 且具有值 null，则显示 null reference exception。
- 枚举函数实例是通过方法引用 C.get enumerator () 求值而得到的。返回的枚举函数存储在临时局部变量内，以下用 E 来表示。嵌套语句不能访问此临时变量。如是 E 是一个 reference-type 且其值为 null，则显示 null reference exception。
- 通过对方法引用 E.move next () 的求值，把这个枚举函数转到下一个元素。
- 如果由 E.move next () 返回的值是 true，则执行以下步骤：
 - 对属性访问 E.current 求值得到当前的枚举函数值，并通过显式转换把这个值转化为循环变量的类型。所得值存储在该循环变量内以便在嵌套语句中它可被访问。
 - 把控制转到该嵌套语句，当控制到达这个嵌套语句的结束点时（可能是从执行一个 continue 语句开始），从上面引用枚举函数的步骤开始，执行另一个 foreach 循环。
- 如果 E.move next () 的返回值是 false，则控制被转到这个 foreach 语句的结束点。

在 foreach 语句的嵌套语句内，break 语句（参见 8.9.1 节）的作用是把控制转到这个 foreach 语句的结束点（这样就结束了此嵌套语句的循环），而 continue 语句的作用则是把控制转到这个嵌套语句的结束点（这样就执行这个 foreach 语句的另一个循环）。

如果一条 foreach 语句是可到达的，则其嵌套语句就是可到达的。同样，如果一条 foreach 语句可到达，则其结束点也可到达。

8.9 jump 语句 (Jump Statements)

jump 语句无条件地转移控制。

```
jump-statement:  
break-statement  
continue-statement  
goto-statement  
return-statement  
throw-statement
```

一条 jump 语句要把控制转移到的地址称为该 jump 语句的 target。当一条 jump 语句出现在一个块内, 而其目标又不在此块内时, 这条 jump 语句就要 exit 这个块。尽管 jump 语句可把控制转到块外, 但它决不能把控制转入一个块。

由于干扰语句 try 的存在, 使得 jump 语句的执行很复杂。当这样的 such 语句缺省时, jump 语句就无条件地把控制从这条 jump 语句转到目标。在干扰语句 try 存在的情况下, 执行要复杂得多。如果 jump 语句中存在一个或多个带有相关 finally 块的 try 块, 则控制就先被转到最里层的 try 语句的 finally 块。当控制到达一个 finally 块的结束点时, 它就被转到相邻的上一级 try 语句的 finally 块。这个过程如此重复直到所有干扰语句 try 的 finally 块都被执行。

在下面的例子中在控制被转到这条 jump 语句的目标之前, 有两条 try 语句的 finally 块被执行:

```
static void F() {  
    while (true) {  
        try {  
            try {  
                Console.WriteLine("Before break");  
                break;  
            }  
            finally {  
                Console.WriteLine("Innermost finally block");  
            }  
        }  
        finally {  
            Console.WriteLine("Outermost finally block");  
        }  
    }  
    Console.WriteLine("After break");  
}
```

8.9.1 break 语句 (The Break Statement)

break 语句存在于最近的上一级 switch、while、do、for 或 foreach 语句中。

```
break-statement:
break;
```

break 语句的目标就是最近的上一级 switch、while、do for 或 foreach 语句的结束点。如果一条 break 语句不包括在 switch、while、do、for 或 foreach 语句之内，则出现编辑错误。

当多个 switch、while、do、for 或 foreach 语句互相嵌套时，break 语句只适用于最里层的语句。要使控制穿过多个嵌套层，必须用 goto 语句。break 语句不含有 finally 块。当一条 break 语句出现在某一个 finally 块内时，这条 break 语句的 target 也必须在同一个 finally 块内，否则，将出现编辑错误。break 语句的执行过程如下：

- 如果这条 break 语句中存在一个或多个带有相关 finally 块的 try 块，则控制就先被转到最里层 try 语句的 finally 块。当控制到达某一 finally 块的结束点时，它就被转到相邻的上一级 try 语句的 finally 块。这个过程如此重复直到所有干扰语句 try 的 finally 块都被执行。
- 控制被转到这条 break 语句的 target。

由于 break 语句无条件地把控制转到其它地方，所以其结束点永远不可到达。

8.9.2 continue 语句 (The Continue Statement)

continue 语句开始执行相邻的上一级 while、do、for 或 foreach 语句的一个新的重复。

```
continue-statement:
continue;
```

continue 语句的目标就是相邻的上一级 while、do、for 或 foreach 语句的嵌套语句的结束点。如果一条 continue 语句不包括在 while、do、for 或 foreach 语句内，则出现编辑错误。

当多个 while、do、for 或 foreach 语句互相嵌套时，continue 语句只适用于最里层的语句。要使控制通过多个嵌套层，必须用一个 goto 语句。continue 语句不能含有 finally 块。当一条 continue 语句出现在一个 finally 块内时，这条 continue 语句的目标必须在同一个 finally 块内，否则，就出现编辑错误。continue 语句的执行过程如下：

- 如果这条 continue 语句中存在一个或多个带有相关 finally 块的 try 块，则控制就首先被转到最里层的 try 语句的 finally 块内。当控制到达一个 finally 块的结束点时，控制就被转到相邻的上一级 try 语句的 finally 块内。这个过程如此重复直到所有干扰语句 try 的 finally 块都被执行。
- 控制被转到这条 continue 语句的目标。

由于 continue 语句无条件地将控制转到其它地方，因此，其结束点永远不可到达。

8.9.3 goto 语句 (The Goto Statement)

goto 语句把控制转到一标号所标记的语句。

```
goto-statement:
    goto identifier ;
    goto case constant-expression ;
    goto default ;
```

goto identified 语句的目标就是已知标号的标记语句。如果在当前的函数成员中不存在已知名字的标号,或者这条 **goto** 语句不在此标号的范围之内,则出现编辑错误。这条规则允许运用一个 **goto** 语句把控制转到嵌套范围之外,但不能转到嵌套范围内。下例中 **goto** 语句的作用是把控制转到一嵌套范围之外:

```
class Test
{
    static void Main(string[] args) {
        int i = 0;
        while (true) {
            Console.WriteLine(i++);
            if (i == 10)
                goto done;
        }
    done:
        Console.WriteLine("Done");
    }
}
```

goto case 语句的目标是最近的上一级 **switch** 语句中的转换部分的语句列表,这个 **switch** 语句包括一给定常量值的 **case** 标号。如果 **goto case** 语句不包括在一 **switch** 语句之内,或其 **constant-expression** 不能被隐式转化为最近的上一级 **switch** 语句的支配类型,或者,最近的上一级 **switch** 语句不包括一给定常量值的 **case** 符号,则出现编辑错误。

goto default 语句的目标就是含有 **default** 标号的最近的上一级 **switch** 语句(参见 8.7.2 节)中切换部分的语句列表。如果这条 **gotodefault** 语句不包括在一 **switch** 语句内,或者,最近的上一级 **switch** 语句不含有 **default** 标号,则出现编辑错误。在 **goto** 语句中,不能有 **finally** 块。当一条 **goto** 语句出现在一 **finally** 块内时,其目标也必须在同一个 **finally** 块内,否则,出现编辑错误。**goto** 语句的执行过程如下:

- 如果一个 **goto** 语句中存在一或多个与 **finally** 有关的 **try** 块。当控制到达这个 **finally** 块的结束点时,就被转到相邻的上一级 **try** 语句的 **finally** 块。这个过程如此重复直到所有干扰语句 **try** 的 **finally** 块都被执行。
 - 控制被转到这条 **goto** 语句的目标。
- 由于 **goto** 语句无条件地把控制转到其它地方,因此,其结束点永远不可到达。

8.9.4 return 语句 (The Return Statement)

return 语句把控制返回到使用该 **return** 语句的函数成员。

```
return-statement:
return expression;
```

没有表达式的 `return` 语句只能用在不求值的函数成员内，这些函数成员包括返回类型为 `void` 的一个方法、属性或索引的 `set` 访问函数、构造函数或析构函数。

带有表达式的 `return` 语句只能被用在求值的函数成员内，这些函数成员指的是：返回类型为 `non-void` 的方法。属性或索引的 `get` 访问函数或用户自定义操作符。从表达式类型到包含它的函数成员的返回类型必须存在一个隐式转换。`return` 语句出现在 `finally` 块内是错误的。

`return` 语句的执行过程如下：

- 如果一条 `return` 语句指定一个表达式，对这个表达式求值并通过隐式转换把所得值转化为包含它的函数成员的返回类型。转换的结果就是返回到调用者的值。
- 如果一条 `return` 语句包括在一或多个具有相关 `finally` 块的 `try` 块内，则控制就首先被转到最里层的 `try` 语句的 `finally` 块内。当控制到达一个 `finally` 块的结束点时，它就被转到相邻的上一级 `try` 语句的 `finally` 块内。这个过程如此重复直到所有的上级 `try` 语句的 `finally` 块都被执行。
- 控制被转到执行它的函数成员。

由于 `return` 语句无条件地把控制转到其它地方，因此，其结束点永远不可到达。

8.9.5 throw 语句 (The Throw Statement)

`throw` 语句的作用是抛出异常：

```
throw-statement:
throw expression;
```

带有表达式的 `throw` 语句抛出由对这个表达式求值而产生的异常。这个表达式必须表示类型 `system exception` 或从 `system exception` 得到的一个类类型的值。如果对这个表达式的求值产生 `null`，则显示 `null reference exception`。

不带表达式的 `throw` 语句只能被用在 `catch` 块内。它重新抛出一个当前正在被这个 `catch` 块操作的异常。由于 `throw` 语句无条件地把控制转到其它地方，因此，其结束点永远不可到达。当异常被抛出时，控制就被转到能处理这个异常的 `try` 语句的第一个 `catch` 句。从正在被抛出的异常点到把控制转到一个合适的异常处理点的操作过程被称为 `exception propagation`。异常的传递包括对下列步骤的重复求值直至找到与这个异常相匹配的 `catch` 句。在这里，`throw point` 最初指的是异常被抛出的地点。

- 在当前的函数成员中，每一个含有抛出点的 `try` 语句都要被检查，对于每一个开始于最里层结束于最外层的 `try` 语句的语句 `s` 来说，按下列步骤求值：
 - 如果 `s` 的 `try` 块含有抛出点且 `s` 具有一或多个 `catch` 句，则这些 `catch` 句都按照它们出现的顺序被检查以便为这个异常找一个合适的处理者。通常认为第一个 `catch` 句就是一个匹配，它指定这个异常类型或其基本类型。一般的 `catch` 句与任何异常类型都匹配。如果一个合适的 `catch` 句被找到，则这个异常传递也就结束了，这时，控制被转到那个 `catch` 句的块中。

- 否则, 如果 `s` 的 `try` 块或 `catch` 块包括抛出点, 且 `s` 具有一个 `finally` 块, 则控制就被转到这个 `finally` 块。如果这个 `finally` 块抛出另一个异常, 则当前的异常就被终止。否则, 当控制到达这个 `finally` 块的结束点时, 当前异常仍然继续进行。
- 如果在当前的函数成员引用中找不到一个异常处理者, 则这个函数成员引用就被终止。对于这个函数成员的调用者来说, 上面的步骤又被重复执行, 这个函数成员带有一个与它被引用时所在的语句一致的抛出点。
- 如果异常处理过程的停止终止当前过程或处理中的所有函数成员引用, 表明这个过程或处理已没有这个异常的处理者, 则这个过程或处理本身就以执行定义的方式被终止。

8.10 try 语句 (The Try Statement)

`try` 语句为捕获在一个块执行期间发生的异常提供了一种方法。而且, 这个 `try` 语句还能指定一个代理块, 这个块在控制离开 `try` 语句时也被执行。

```
try-statement:
    try block catch-clauses
    try block finally-clause
    try block catch-clauses finally-clause

catch-clauses:
    specific-catch-clauses general-catch-clause_opt
    specific-catch-clauses_opt general-catch-clause

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause

specific-catch-clause:
    catch ( class-type identifier_opt ) block

general-catch-clause:
    catch block

finally-clause:
    finally block
```

`try` 语句有三种可能形式:

- 带有一或多个 `catch` 块的 `try` 块。
- 带有一 `finally` 块的 `try` 块。
- 带有一或多个 `catch` 块其后又有一个 `finally` 块的 `try` 块。

当一个 `catch` 句指定一个 `class-type` 时, 这个类型必须是 `system exception` 或从 `system exception` 得到的一个类型。当一个 `catch` 句既指定一个 `class-type`, 又指定一个 `identifier` 时, 一给定名字及类型的 `exception variable` 就被声明。其异常变量与其范围超过这个 `catch` 块的局部变量一致。在这个 `catch` 块的执行过程中, 该异常变量就表示当前正在被处理的异常。为

方便明确赋值检查，认为此异常变量在其整个范围内都是被明确赋值的。

除非一个 `catch` 句包括一个异常变量名，否则，在这个 `catch` 块中不可能访问异常对象。

既没有指定的异常类型，也没有指定的异常变量名的 `catch` 句称为普通 `catch` 句。一个 `try` 语句只能有一个普通 `catch` 句，且如果有一个存在，它必须是最后的 `catch` 句。

尽管 `throw` 语句只能抛出类型 `system exception` 或由 `system exception` 派生的一个类型的异常，但其它语言不受这条规则的束缚，因此，可以抛出其它类型的异常。普通的 `catch` 句的作用是捕获这样的异常，而不带表达式的 `throw` 语句的作用则是重新抛出它们。

如果一个 `catch` 句指定一个与在前面的 `catch` 句中已被指定的类型相同，或由它派生而来的类型，则将出现错误。因为 `catch` 句是按它们作为异常寻找者时出现的顺序被检查的，如果没有这个限制条件，那么，有可能书写一个不可到达的 `catch` 句。在一 `catch` 块内，不带表达式的 `throw` 语句（参见 8.9.5 节）可重新抛出被这个 `catch` 块所捕获的异常。对异常变量的赋值不改变这个重新被抛出的异常。

在下面的例子中方法 `f` 捕获一个异常，并把有关信息传给控制器，改变这个异常变量，并重新抛出这个异常：

```
class test
{
    static void f() {
        try {
            g();
        }
        catch (exception e) {
            Console.WriteLine("exception in f: " + e.message);
            e = new exception("f");
            throw;           // re-throw
        }
    }

    static void g() {
        throw new exception("g");
    }

    static void main() {
        try {
            f();
        }
        catch (exception e) {
            Console.WriteLine("exception in main: " + e.message);
        }
    }
}
```

这个被重新抛出的异常就是原始异常，因此，程序输出：

```
exception in f: g
exception in main: g
```

break continue 或 **goto** 语句把控制转到 **finally** 块外是错误的。当一个 **break continue** 或 **goto** 语句出现在一个 **finally** 块内时，这条语句的目标也必须在同一个 **finally** 块内，否则，出现编辑错误。在一个 **finally** 块内出现 **return** 语句是错误的。

try 语句的执行过程如下：

- 控制被转到这个 **try** 块。
- 当控制到达这个 **try** 块的终点时：
 - 如果这个 **try** 语句具有一个 **finally** 块，则执行这个 **finally** 块。
 - 控制被转到这个 **try** 语句的结束点。
- 在这个 **try** 块的执行过程中，如果有一个异常传到这个 **try** 语句：
 - 如果有 **catch** 句，则它们将按其为异常寻找合适处理器时出现的顺序被检查。第一个 **catch** 语句就被认为是一个匹配，它指定这个异常类型或其基本类型。普通的 **catch** 句与任何异常类型都匹配。如果一个匹配的 **catch** 句被找到：
 - 如果这个匹配的 **catch** 句声明一个异常变量，这个异常就被赋给此异常变量。
 - 控制被转到这个匹配的 **catch** 块。
 - 当控制到达此 **catch** 块的结束点时：
 - 如果这条 **try** 语句具有一个 **finally** 块，则执行这个块。控制被转到这个 **try** 语句的结束点。
 - 在这个 **catch** 块的执行过程中，如果一个异常被传到这条 **try** 语句：
 - 如果这个 **try** 语句具有一个 **finally** 块，则执行这个块。该异常被传到相邻的上一级 **try** 语句。
 - 如果这个 **try** 语句没有 **catch** 句或没有与该异常匹配的 **catch** 句：
 - 如果这个 **try** 语句具有一个 **finally** 块，则执行这个 **finally** 块。该异常被传到相邻的上一级 **try** 语句。

当控制离开一个 **try** 语句时，**finally** 块的语句仍被执行。不管控制转移是正常执行的结果，还是执行一个 **break continue goto** 或 **return** 语句的结果，或者是把一个异常传到这个 **try** 语句之外的结果，上述情况都是正确的。在 **finally** 块的执行过程中，如果一个异常被抛出，则这个异常就被传到相邻的上一级 **try** 语句。如果其它的异常正在被传送，则那个异常就要丢失。关于异常的传递过程在描述 **throw** 语句时进行了详细的探讨。

- 如果一条 **try** 语句可到达，则这条 **try** 语句的 **try** 块就可到达。
- 如果一个 **try** 语句可到达，则这个 **try** 语句的 **catch** 块也可到达。
- 如果一个 **try** 语句可到达，则这个 **try** 语句的 **finally** 块也可到达。

如果下列条件都符合，则一个 **try** 语句的结束点就是可达到的。

- 这个 **try** 块的结束点可到达或至少一个 **catch** 块的结束点可到达。
- 存在一个 **finally** 块，且这个 **finally** 块的结束点可到达。

8.11 checked 和 unchecked 语句 (The Checked And Unchecked Statements)

checked 和 unchecked 语句的作用是控制整类型算术操作及转换的 overflow checking context。

```
checked-statement:
    checked block

unchecked-statement:
    unchecked block
```

checked 语句使得此 block 中的所有表达式在 checked 上下文中被求值，而 unchecked 语句使得此 block 中的所有表达式在 unchecked 上下文中被求值。除在块中外，checked 和 unchecked 语句分别与 checked 和 unchecked 操作符（参见 7.5.12 节）完全等同。

8.12 lock 语句 (The Lock Statement)

lock 语句为一给定对象加上互不包括的锁定，执行一个语句，然后释放这个锁定。

lock-statement;

lock (expression) embedded-statement

lock 语句的表达式必须表示 reference-type 的一个值。且对 lock 语句的表达式来说，不可能执行隐式装箱转换，因此，lock 表达式表示 value-type 的值是错误的。形式为 lock (x) ... 的一个 lock 语句与下面的语句完全相等，这里，x 是 reference-type 的一个表达式。

```
{
    System.Threading.Monitor.Enter(x);
    try {
        ...
    }
    finally {
        System.Threading.Monitor.Exit(x);
    }
}
```

除非 x 只能被求值一次，否则，类 system threading monitor 的方法 enter 和 exit 的确切功能在执行过程中可定义。一个类的 system type 可用作这个类静态方法的互不包括锁定。例如：

```
class Cache
{
    public static void Add(object x) {
        lock (typeof(Cache)) {
            ...
        }
    }
}
```

```

public static void Remove(object x) {
    lock (typeof(Cache)) {
        ...
    }
}
}

```

8.13 using 语句 (The Using Statement)

using 语句得到一或多个援助, 执行一个语句, 然后除去这个援助。

```

using-statement:
    using ( resource-acquisition ) embedded-statement

resource-acquisition:
    local-variable-declaration
    expression

```

一个 resource 就是执行 system idisposable 的一个类或结构, 它包括一个名字为 dispose 的不带参数的方法。正在使用援助的代码可以调用 dispose 以示这个援助已不再需要。如果 dispose 不被调用, 那么, 自动处理最终就会以收集垃圾后的结果而出现。

如果 resource-acquisition 的形式是 local-variable-declaration 那么, 这个 local-variable-declaration 的类型就必须是 system idisposable 或可被隐式转化为 system idisposable 的类型。如果 resource-acquisition 的形式是 expression, 那么, 这个表达式必须是 system idisposable 或可被隐式转化为 system idisposable 的类型。在 resource-acquisition 中声明的局部变量是只读的, 且包括一个初始化。

一条 using 语句包括三部分: acquisition, usage 及 disposal。援助的运用发生在含有 finally 分句的 try 语句内。这里的分句 finally 的作用是取消这个援助。如果得到的援助是 null, 则不调用 dispose, 且也无异常抛出。

例如, 形式为

```

using (R r1 = new R ()) {
    r1.F();
}

```

的 using 语句与下面的语句完全相等:

```

R r1 = new R();
try {
    r1.F();
}
finally {
    if (r1 != null) ((IDisposable)r1).Dispose();
}

```

一个 resource-acquisition 可能得到一给定类型的多个援助。这与被嵌套的 using 语句相同。例如, 下例的 using 语句:

```
using (R r1 = new R(), r2 = new R()) {  
    r1.F();  
    r2.F();  
}
```

与下面的语句完全一致:

```
using (R r1 = new R())  
using (R r2 = new R()) {  
    r1.F();  
    r2.F();  
}
```

从广义上来讲, 它又与下面的例子一致:

```
R r1 = new R();  
try {  
    R r2 = new R();  
    try {  
        r1.F();  
        r2.F();  
    }  
    finally {  
        if (r2 != null) ((IDisposable)r2).Dispose();  
    }  
}  
finally {  
    if (r1 != null) ((IDisposable)r1).Dispose();  
}
```

第9章 名字空间

C#程序是用名字空间组织起来的。名字空间既用作一个程序的“内部”组织系统，又用作“外部”组织系统——提供暴露给其它程序的程序元素的一种方式。为便于使用名字空间，提供了使用指令。

9.1 编译单元 (Compilation Units)

编译单元 (compilation-unit) 定义源文档的总体结构。一个编译单元包括零或多个带有零或多个 namespace-member-declarations 的 using-directives。

```
compilation-unit:  
    using-directivesopt attributesopt namespace-member-declarationsopt
```

一个 C#程序是由一或多个编译单元组成的，每一个编译单元都包含在一独立的源文档中。编译 C#程序时，所有这些编译单元同时被操作。这样，编译单元就可能循环依赖。一个编译单元的 using-directives 影响其 attributes 及 namespace-member-declarations，但不影响其它编译单元。编译单元的 attributes 允许目标集合及组件的属性说明。集合和组件就相当于类型的有形容器。一个集合可包括几个自然独立的组件。

程序的每一个编译单元的 namespace-member-declarations 都为 一个称为总名字空间的名字空间提供成员。例如：

```
File A.cs:  
    class A {}
```

```
File B.cs:  
    class B {}
```

两个编译单元组成一个总名字空间，这个例子声明两个全权名字分别为 A 和 B 的类。由于这两个编译单元提供相同的声明空间，因此，如果每一个都含有具有相同名字成员的声明，则出现错误。

9.2 名字空间声明 (Namespace Declarations)

一个名字空间声明 (namespace-declaration) 包括关键词 namespace，其后是一个名字空间名字及主体，然后是一可选择的分号。

```
namespace-declaration:  
    namespace qualified-identifier namespace-body ;
```

```

qualified-identifier:
    identifier
    qualified-identifier . identifier

namespace-body:
{ using-directivesopt namespace-member-declarationsopt }

```

namespace-declaration 可能作为 **compilation-unit** 内的一个最外层的声明而发生，也可能作为另一个 **namespace-declaration** 内的成员声明而发生。当 **namespace-declaration** 作为前者而发生时，这个名字空间就是总名字空间的一个成员。当它作为后者而发生时，里层名字空间就是外层名字空间的一个成员。在两种情况下，名字空间的名字在包含它的名字空间中必须是唯一的。

名字空间是绝对 **public** 的，且名字空间的声明不包括任何访问修改函数。在一个 **namespace-body** 内，可选的 **using-directives** 输入其它名字空间及类型的名字，并允许它们直接被引用而不必使用权名。可选的 **namespace-member-declarations** 为这个名字空间的声明提供成员。注意，所有的 **using-directives** 都必须出现在成员声明之前。

namespace-declaration 的 **qualified-identifier** 可以是一个标识符或一系列由“.”隔开的标识符。后一种形式允许一个程序在不编辑嵌套名字空间声明的情况下定义一个嵌套名字空间。例如：

```

namespace N1.N2
{
    class A {}

    class B {}
}

```

从语法上来讲，上例与下面的例子是一致的：

```

namespace N1
{
    namespace N2
    {
        class A {}

        class B {}
    }
}

```

名字空间是开放式结束的，且具有相同全权名字的名字空间声明产生相同的声明空间（参见 3.3 节），在下面的例子中：

```

namespace N1.N2
{
    class A {}
}

```

```
namespace N1.N2
{
    class B {}
}
```

以上两个名字空间声明产生相同的声明空间，在这个例子中声明两个全权名字分别为 N1 N2 A 和 N1 N2 B 的类。由于这两个声明产生相同的声明空间，因此，如果每一个都包括具有相同名字成员的声明，那么，就是错误的。

9.3 使用指令 (Using Directives)

使用指令 (using directives) 使得名字空间及在其它名字空间中定义的类型的使用更方便。使用指令影响 namespace-or-type-names (参见 3.8 节) 及 simple-names (参见 7.5.2 节) 的名字确定过程，但是，与声明不同，使用指令不给编译单元或使用它们名字空间的潜在声明空间提供新成员。

```
using-directives:
    using-directive
    using-directives using-directive

using-directive:
    using-alias-directive
    using-namespace-directive
```

using-alias-directive (参见 9.3.1 节) 为名字空间或类型提供参考别名。using-namespace-directive (参见 9.3.2 节) 输入名字空间的类型成员。using-directive 的范围延伸至直接包含它的编译单元或名字空间主体的 namespace-member-declarations。但 using-directive 不包括与其同级的 using-directives。因此，同级的 using-directives 之间不互相影响，且它们的书写顺序也不重要。

9.3.1 别名指令使用 (Using Alias Directives)

using-alias-directive 输入一个作为某名字空间或类型的别名的标识符，这个名字空间或类型在最近的上一级编译单元或名字空间主体内。

```
using-alias-directive:
using identifier=namespace-or-type-name;
```

在包含 using-alias-directive 的编译单元或名字空间内的成员声明中，由 using-alias-directive 输入的标识符的作用是引用给定的名字空间或类型。例如：

```
namespace N1.N2
{
    class A {}
}
```



```

    }
    namespace N3
    {
        using A = N1.N2.A;
        class B: A {}
    }

```

这里，在名字空间 N3 的成员声明内，A 是 N1.N2.A 的一个别名，因此，类 N3B 是由类 N1.N2.A 派生而来的。通过建立 N1.N2 的一个别名 R，并引用 RA 可得到同样的效果。

```

namespace N3
{
    using R = N1.N2;

    class B: R.A {}
}

```

一个 `using-alias-directive` 的 `identifier` 在直接包含这个 `using-alias-directive` 的编译单元或名字空间的声明空间内必须是唯一的。例如：

```

namespace N3
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;    // Error, A already exists
}

```

这里，N3 已经含有了一个成员 A，因此 `using-alias-directive` 不能再用那个标识符，否则就是错误的。在同一个编译单元或名字空间主体内的两个或多个 `using-alias-directive` 声明具有相同的别名也同样是错误的。

`using-alias-directive` 使得别名可在特定的编译单元或名字空间内取得，但它不给潜在的声明空间提供任何新成员。也就是说，`using-alias-directive` 不可能转移，它只能影响它所在的编译单元或名字空间主体。在下例中：

```

namespace N3
{
    using R = N1.N2;
}

namespace N3
{
    class B: R.A {}    // Error, R unknown
}

```

引入 `r` 的 `using-alias-directive` 的范围只延伸至包含它的名字空间主体的成员声明,因此在第二个名字空间声明中, `r` 是未知的。然而,如果把 `using-alias-directive` 放在包含它的编译单元中就会使得这个别名在两个名字空间声明中都能得到与常规成员相同,由 `using-alias-directive` 所引入的名字也会被嵌套范围内具有相似名字的成员所隐藏:

```
using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}
```

在下例中:

```
using R = N1.N2;

namespace N3
{
    class R {}

    class B: R.A {}    // Error, R has no member A
}
```

在 `B` 的声明过程中,对 `R.A` 的引用出现错误,因为 `R` 指的是 `N3R` 而不是 `N1N2`。

`using-alias-directive` 的书写顺序并不重要,被 `using-alias-directive` 所引用的 `name space-or-type-name` 的取消既不受 `using-alias-directive` 本身的影响,也不受直接包含它的编译单元或名字空间主体中其它 `using-directives` 的影响。也就是说,在取消 `using-alias-directive` 的 `name space-or-type-name` 时,就相当于相邻的上一级编译单元或名字空间主体不含有 `using-directives`。在下例中最后的 `using-alias-directive` 是错误的,因为它不受第一个 `using-alias-directive` 的影响:

```
namespace N1.N2 {}

namespace N3
{
    using R1 = N1;           // OK

    using R2 = N1.N2;        // OK

    using R3 = R1.N2;        // Error, R1 unknown
}
```

`using-alias-directive` 可以为任何名字空间或类型建立一个别名,包括它所存在的名字空间及任何嵌套在那个名字空间内的名字空间或类型。通过别名访问名字空间或类型与通过其声

明的名字访问这个名字空间或类型的结果是一样的,也就是说,在下例中名字 N1 N2 A R1 N2 A 及 R2 A 是完全一致的,都指全权名字为 N1 N2 A 的类:

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;           // refers to N1.N2.A
        R1.N2.A b;           // refers to N1.N2.A
        R2.A c;              // refers to N1.N2.A
    }
}
```

9.3.2 名字空间指令使用 (Using Namespace Directives)

using-namespace-directive 把包含在名字空间内的类型输入到相邻的上一级编译单元或名字空间主体内,使得每种类型的标识符不必经过授权都可被应用。

```
using-namespace-directive;
using namespace;
```

在含有 using-namespace-directive 的编译单元或名字空间主体内的成员声明过程中,包含在给定名字空间内的类型可直接被引用。例如:

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1.N2;

    class B: A {}
}
```

这里,在名字空间 N3 的成员中, N1 N2 的类型成员可直接得到,因此,类 N3 B 是由类

N1 N2A 派生而来的。

`using-namespace-directive` 输入包含在给定名字空间内的类型，但它不输入嵌套名字空间。在下例中 `using-namespace-directive` 输入包含在 N1 内的类型，但并不输入嵌套在 N1 内的名字空间：

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;

    class B: N2.A {}           // Error, N2 unknown
}
```

因此，在 B 的声明过程中对 N2A 的引用是错误的，因为命名为 N2 的成员不在其范围内。

与 `using-alias-directive` 不同，`using-namespace-directive` 可以输入这样的类型，即其标识符在上一级编译单元或名字空间主体内已被定义的类型。实际上，由 `using-namespace-directive` 输入的名字被包含它的编译单元或名字空间主体中具有相似名字的成员所隐藏。例如：

```
namespace N1.N2
{
    class A {}

    class B {}
}

namespace N3
{
    using N1.N2;

    class A {}
}
```

这里，在名字空间 N3 的成员声明过程中，A 指的是 N3 A 而不是 N1N2A。

当由同一个编译单元或名字空间主体内的 `using-namespace-directive` 输入的多个名字空间含有具有相同名字的类型时，对那个名字的引用就是不明确的。在下例中 N1 和 N2 都含有一个成员 A，而 N3 中输入了两个，因此在 N3 中对 A 的引用就是错误的：

```
namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}
```

```

class A {}
}
namespace N3
{
using N1;
using N2;
class B: A {}           // Error, A is ambiguous
}

```

在这种情况下，这种冲突可通过对 A 引用的限制或引入一个能挑出一特定 A 的 `using-alias-directive` 而解决。例如：

```

namespace N3
{
using N1;

using N2;

using A = N1.A;

class B: A {}           // A means N1.A
}

```

与 `using-alias-directive` 相同，`using-namespace-directive` 不给这个编译单元或名字空间的潜在声明空间提供任何新成员，而只影响它所在的编译单元或名字空间主体。

被 `using-namespace-directive` 所引用的 `namespace-name` 的确定方式与被 `using-alias-directive` 所引用的 `namespace-or-type-name` 的确定方式相同。因此，在同一个编译单元或名字空间主体内的 `using-namespace-directive` 之间不互相影响且可以按任何顺序书写。

9.4 名字空间成员 (Namespace Members)

`namespace-member-declaration` 或者是 `namespace-declaration` (参见 9.2 节)，或者是 `type-declaration` (参见 9.5 节)。

```

namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
    namespace-declaration
    type-declaration

```

编译单元或名字空间主体可含有 `namespace-member-declarations`，且这类声明给包含它的编译单元或名字空间主体的潜在声明空间提供新的成员。

9.5 类型声明 (Type Declarations)

`type-declaration` 或者是 `class-declaration`、`struct-declaration`、`interface-declaration`、`enum-declaration` 或一个 `delegate-declaration`。

```
type-declaration:  
    class-declaration  
    struct-declaration  
    interface-declaration  
    enum-declaration  
    delegate-declaration
```

`type-declaration` 可以作为编译单元内最外层的声明而出现，也可以作为名字空间、类或结构内的一个成员声明而出现。

当类型 `F` 的声明作为编译单元内的最外层声明而出现时，这个新声明类型的全权名字就是 `T`。当类型 `T` 的声明发生在一名字空间、类或结构内时，这个新声明类型的全权名字就是 `NT`，这里 `N` 就是这个名字空间、类或结构的全权名字。在类或结构内声明的类型称为嵌套类型（参见 10.2.6 节）。

类型声明所允许的访问修改函数及缺省访问取决于这个声明发生（参见 3.5.1 节）的上下文：

- 在编译单元或名字空间内声明的类型可具有 `public` 或 `internal` 访问。缺省时就是 `internal` 访问。
- 在类中声明的类型可具有 `public`、`protected internal`、`protected`、`internal` 或 `private` 访问。缺省时为 `private` 访问。
- 在结构中声明的类型可具有 `public`、`internal` 或 `private` 访问。缺省时为 `private` 访问。

第 10 章 类

一个类就是一个数据结构，它包括数据成员（常量、域和事件）、函数成员（方法、属性、索引、操作符构造函数和析构函数）及嵌套类型。类类型支持继承，通过继承，派生类可延伸并可定义一个基类。

10.1 类声明（Class Declarations）

一个 `class-declaration` 就是声明一个新类的 `type-declaration`（参见 9.5 节）

```
class-declaration:  
attributesopt class-modifiersopt class identifier class-baseopt  
class-body ;opt
```

`class-declaration` 包括 `attributes`（参见 17 节）的一个可选择的集合，然后是关键词 `class` 及作为这个类名字的 `identifier`，然后是一个可选择的 `class-base` 说明表（参见 10.1.2 节），其后是一个 `class-body`（参见 10.1.3 节），最后可选择地跟一个分号。

10.1.1 类修改函数（Class Modifiers）

一个 `class-declaration` 可选择地包括一系列类修改函数：

```
class-modifiers:  
class-modifier  
class-modifiers class-modifier  
class-modifier:  
new  
public  
protected  
internal  
private  
abstract  
sealed
```

在类声明中，不允许同一个修改函数出现多次，否则就是错误的。

修改函数 `new` 只允许出现在嵌套类中。它表明这个类隐藏一个同名的继承成员，如 10.2.2 节所述。修改函数 `public`，`protected`，`internal` 及 `private` 控制类的可访问性。根据发生类声明的上下文，这些修改函数中有一些可能不被允许（参见 3.5.1 节）。修改函数 `abstract` 和 `sealed` 将在下面几部分加以说明。

10.1.1.1 抽象类 (Abstract Classes)

`abstract` 修改函数表明一个类是不完整的, 只能作为其它类的一个基类。抽象类与非抽象类的不同之处有以下几个方面:

- 抽象类无法直接用例子说明, 且在抽象类中不能使用操作符 `new`。尽管可能有编译期类型是抽象的变量和值, 但这样的变量和值一定或者是 `null`, 或者含有对从这个抽象类型派生的非抽象类的实例的引用。
- 抽象类允许 (但不需要) 含有抽象成员。
- 抽象类不能被密封。

当一非抽象类是从一抽象类派生而来时, 这个非抽象类一定包括所有被继承抽象成员的实际执行。这样的执行是通过使抽象成员无效而得到的。在下例中:

```
abstract class a
{
    public abstract void f();
}

abstract class b: a
{
    public void g() {}
}

class c: b
{
    public override void f() {
        // actual implementation of f
    }
}
```

抽象类 `a` 引入一个抽象方法 `f`。类 `b` 引入一个额外的方法 `g`, 但不提供 `f` 的执行。因此, `b` 也必须被声明为抽象。类 `c` 使 `f` 无效 (越过 `f`) 并提供一个实际执行。由于在 `c` 中没有明显的抽象成员, 因此, 允许 `c` (但不需要) 是非抽象的。

10.1.1.2 封装类 (Sealed Classes)

修改函数 `sealed` 的作用是阻止从一个类派生。如果一个封装类被指定为另一个类的基类, 则错误发生。封装类也不能是一个抽象类。修改函数 `sealed` 的作用首先是阻止无目的的派生, 但它也能使一定的运行期最佳化。尤其是, 因为已知一个封装类不可能具有任何派生类, 因此, 就可把封装类实例中的虚拟函数成员引用转化为非虚拟引用。

10.1.2 类基本说明 (Class Base Specification)

类声明可能包括一个 `class-base` 说明, 这个 `class-base` 说明定义该类的直属基类及其执行接口。


```

class-base:
    : class-type
    : interface-type-list
    : class-type , interface-type-list

interface-type-list:
    interface-type
    interface-type-list , interface-type

```

10.1.2.1 基类 (Base Classes)

当 `class-base` 包括一个 `class-type` 时, 这个 `class-type` 就指定正在被声明的类的直属基类。如果一个类的声明过程没有 `class-base`, 或者这个 `class-base` 只列举接口类型, 那么, 这个直接的基类就被假定为 `object`。一个类从其直属基类中继承成员, 如 10.2.1 节所述。

在下例中:

```

class a {}

class b: a{}

```

这里, 类 `a` 是 `b` 的直属基类, 而 `b` 是从 `a` 派生而来的。由于 `a` 并未明确指定一个直属基类, 因此, 它的直属基类默认为 `object`。类类型的直属基类必须至少与其本身 (参见 3.5.4 节) 的可访问性相同。例如, `public` 类从 `private` 或 `internal` 类派生而来是错误的。

类类型的直属基类决不能是下列任何一种类型:

`system.array`, `system.delegate`, `system.enum` 或 `system.value type`。

一个类的基类就是其直属基类及这些直属基类的基类。也就是说, 基类的集合就是一个链式直属基类的范围。根据上面的例子, `b` 的基类就是 `a` 和 `object`。除类 `object` 外, 每一个类都只有一个直属基类。类 `object` 没有直属基类, 是所有其它类的最小基类。

当一个类 `b` 是从类 `a` 派生而来时, 认为 `a` 取决于 `b` 是错误的。一个类 `directly depends on` 它的直属基类 (如果有) 并 `directly depends on` 直接嵌套它的类 (如果有)。决定类的完整的集合就是这个逐级 `directly depends on` 关系的可传递的范围。下面的例子:

```

class A: B {}

class B: C {}

class C: A {}

```

是错误的, 因为这些类循环地依赖于它们自己。同样, 下例

```

class A: B.C {}

class B: A
{
    public class C {}
}

```

是错误的, 因为 `A` 依赖于 `B.C` (它的直属基类), 后者由依赖于 `B` (直接包含它的类), 而 `B` 又循环地依赖于 `A`。

注意，一个类不能依赖于它的嵌套类，下例中

```
class A
{
    class B: A {}
}
```

B 取决于 A (因为 A 既是它的直属基类，又是直接包括它的类)，但是 A 不取决于 B (因为 B 既不是一个基类，也不包括 A)。因此，上面的例子是有效的。

不可能从 sealed 类派生。在下例中：

```
sealed class A {}
class B: A {} // Error, cannot derive from a sealed class
```

类 B 是错误的，因为它试图从 SEALED 类 A 派生而来。

10.1.2.2 接口执行 (Interface Implementations)

对 class-based 的说明可能包括接口类型的一个列表，在这种情况下，就说这个类执行给定的接口类型。接口执行将在 13.4 节中进一步讨论。

10.1.3 类主体 (Class Body)

一个类的 class-body 定义这个类的成员。

```
Class-body:
    {class-member-declarationsopt}
```

10.2 类成员 (Class Member)

一个类的成员包括由其 class-member-declarations 引入的成员及从其直属基类继承的成员。

```
class-member-declarations:
    class-member-declaration
    class-member-declarations class-member-declaration

class-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    destructor-declaration
    static-constructor-declaration
    type-declaration
```

一个类的成员可分为以下几种:

- 常量, 表示与该类有关的常量值 (参见 10.3 节)。
- 域, 即这个类的变量 (参见 10.4 节)。
- 方法, 执行计算及可被这个类执行的操作 (参见 10.6 节)。
- 属性, 说明已知名的属性及与读和写那些属性有关的操作 (参见 10.6 节)。
- 事件, 说明由这个类产生的通知 (参见 10.7 节)。
- 索引, 允许这个类的实例以与数组相同的方式被索引 (参见 10.8 节)。
- 操作符, 定义可用于这个类的实例的表达式操作符 (参见 10.9 节)。
- 实例构造函数, 执行初始化这个类的实例时所要求的操作 (参见 10.10 节)。
- 析构函数, 执行这个类的实例被永久抛弃之前的操作 (参见 10.11 节)。
- 类型, 表示这个类的地址的类型 (参见 9.5 节)。

含有可执行代码的成员统称为这个类的 **function members**。一个类的函数成员就是这个类的方法, 属性、索引、操作符、构造函数及析构函数。

一个 **class-declaration** 建立一个新的声明空间 (参见 3.3 节), 且直接包含在 **class-declaration** 内的 **class-member-declarations** 把新的成员引入这个声明空间。下列规则适用于 **class-member-declarations**:

构造函数和析构函数必须与直接包含它们的类具有相同的名字。所有其它成员的名字必须与直接包含它们的类的名字不同。

- 常量、域、属性、事件或类型的名字必须与在同一个类中声明的所有其它成员的名字不同。
- 方法的名字必须与在同一类中声明的所有非方法成员的名字不同。另外, 在同一类中声明的方法的签名 (参见 3.6 节) 必须互不相同。
- 在同一类中声明的构造函数的用法说明必须彼此不同。
- 在同一类中声明的索引函数的用法说明必须彼此不同。
- 在同一类声明操作符函数的用法说明必须彼此不同。

一个类被继承的成员 (参见 10.2.1 节) 不是该类声明空间的一部分。因此, 允许一个派生的类声明一个与被继承成员具有相同的名字或签名的成员 (这实际上是隐藏了这个被继承成员)。

10.2.1 继承 (Inheritance)

一个类 **inherits** 其直属基类的成员。继承指的是一个类固定包括其直属基类的所有成员, 其基类的构造函数和析构函数除外。继承的主要特点是:

- 继承可传递, 如果 **c** 是由 **b** 派生而来的, 而 **b** 又是由 **a** 派生而来的, 则 **c** 既继承在 **a** 中声明的成员, 又继承在 **b** 中声明的成员。
- 派生类 **extends** 其直属基类。派生类可向其继承的类中增加新成员, 但它不能消除被继承成员的定义。
- 构造函数和析构函数不能被继承, 但所有其它成员, 不管其声明的可访问性如何都可被继承。然而, 根据它们声明的可访问性, 被继承的成员在派生类中不可被访问。
- 派生类可以通过声明新的具有相同名字或签名的成员隐藏 (参见 3.7.1.2 节) 继承成

员。对继承成员的隐藏并不等于取消了那个成员——只是使那个成员在派生类中不可被访问。

- 一个类的实例包括所有在这个类及其基类中声明的实例域的一个拷贝，且存在一个从派生的类类型到其任何基类类型的隐式转换（参见 6.1.4 节）。因此，对派生类实例的引用也可以看作是对基类实例的引用。
- 一个类可以声明虚拟的方法，属性和索引，而派生类则可以越过这些函数成员的执行。这使得类的功能呈现多样性，其中，由一个函数成员引用所执行的操作随着引用这个函数成员的实例运行期类型的变化而变化。

10.2.2 new 修改函数 (The New Modifier)

一个 `class-member-declaration` 允许声明一个与继承成员具有相同名字或签名的成员。当这种现象发生时，就说派生的类成员隐藏了基类成员。隐藏继承成员并非一个错误，但它却能使编译发出警告。为消除警告，可在派生类成员声明过程中增加一个 `new` 修改函数以表明派生的成员要隐藏基类成员。这个主题在 3.7.1.2 中有进一步探讨。

如果 `new` 修改函数存在十一个隐藏继承成员的声明中，则将出现警告。这个警告可通过去掉 `new` 修改函数而消除。在同一声明中引用 `new` 和 `override` 修改函数是错误的。

10.2.3 访问修改函数 (Access Modifiers)

一个 `class-member-declaration` 可以具有声明的可访问性（参见 3.5.1 节）五种可能类型中的任何一种，`public`、`protected internal`、`protected`、`internal` 或 `private`。除 `protected internal` 组合外，不能指定多于一个的访问修改函数，否则，就是错误的。如果某一个 `class-member-declaration` 不包含任何访问修改函数时，则声明的可访问性就默认为是 `private`。

10.2.4 constituent 类型 (Constituent Types)。

成员声明过程中所引用的类型称为该成员的 `constituent` 类型。

`constituent` 类型有：常量、域、属性、事件或索引的类型，方法或操作符的返回类型及方法、索引、操作符或构造函数的参数类型。一个成员的 `constituent` 类型的可访问性必须至少与这个成员本身相同（参见 3.5.4 节）。

10.2.5 静态和实例成员 (Static And Instance Members)

类的成员或者是 `static members` 或者是 `instance members`。一般地说，把静态成员归类为类，实例成员归类为对象（类的实例）是有意义的。

当一个域、方法、属性、事件、操作符或构造函数声明含有 `static` 修改函数时，它就声明具有以下特征一个静态成员：

- 当静态成员在式子 `e · m` 的一个 `member-access` 中被引用时，`e` 必须表示类型。如果 `e` 表示实例，就是错误的。

- 静态域只标识存储空间。不管一个类建立多少实例，只有一个静态域的拷贝。
- 静态函数成员（方法、属性、索引、操作符或构造函数）在具体实例中起作用，在静态函数成员中引用 `this` 也是错误的。当一个域、方法、属性、事件、索引、构造函数或析构函数声明不含有 `static` 修改函数时，它就声明一个实例成员。实例成员有时称为非静态成员。实例成员具有下列特征：
 - 当实例成员在式子 `e.m` 的 `member-access` 中被引用时，`e` 必须表示一个实例。`e` 表示一个类型是错误的。
 - 一个类的每一个实例都包括这个类的所有实例域的一个独立的拷贝。
 - 实例函数成员（方法、属性、访问函数、索引访问函数、构造函数或析构函数）在类的给定实例中起作用，且这个实例可被当作 `this` 访问。

下例表明了访问静态和实例成员规则：

```
class Test
{
    int x;
    static int y;
    void F() {
        x = 1;           // Ok, same as this.x = 1
        y = 1;           // Ok, same as Test.y = 1
    }
    static void G() {
        x = 1;           // Error, cannot access this.x
        y = 1;           // Ok, same as Test.y = 1
    }
    static void Main() {
        Test t = new Test();
        t.x = 1;         // Ok
        t.y = 1;         // Error, cannot access static member through instance
        Test.x = 1;      // Error, cannot access instance member through type
        Test.y = 1;      // Ok
    }
}
```

方法 `f` 表明，在实例函数成员中，`simple-name` 既可访问实例成员，又可访问静态成员。方法 `g` 表明，在静态函数成员中，通过 `simple-name` 访问实例成员是错误的。方法 `main` 表明，在 `member-access` 中，实例成员必须通过实例被访问，而静态成员必须通过类型被访问。

10.3 常量 (Constants)

一个 `constant` 就是一个表示常量值的类成员；一个可在编译期被计算的值。`constant-declaration` 引入给定类型的一个或多个常量。

```

constant-declaration:
    attributesopt constant-modifiersopt const type
    constant-declarators ;

constant-modifiers:
    constant-modifier
    constant-modifiers constant-modifier

constant-modifier:
    new
    public
    protected
    internal
    private

constant-declarators:
    constant-declarator
    constant-declarators , constant-declarator

constant-declarator:
    identifier = constant-expression

```

constant-declaration 可能包括一系列 attributes, 一个 new 修改函数及四个访问修改函数的有效组合。属性及修改函数适用于所有由 constant-declaration 声明的成员。即使常量被认为是静态成员, constant-declaration 既不需要也不允许有 static 修改函数。

constant-declaration 的 type 规定由此声明所引入的成员的类型。这个类型后跟一个 constant-declarators 的列表, 其中, 每一个 constant-declarator 引入一个新成员。一个 constant-declarator 包括作为这个成员名字的 identifier, 然后是符号 "=", 最后是给出这个成员值的一个 constant-expression。

常量声明中所指定的 type 必须是: sbyte、byte、short、ushort、int、uint、long、char、float、double、decimal、bool、string 一个 enum-type 或 reference-type。每一个 constant-expression 必须计算出目标类型或可通过隐式转换转化为目标类型的类型值。一个常量的 type 的可访问性必须至少与这个常量本身相同。常量本身可参与 constant-expression。因此, 常量可被用于需 constant-expression 的任何结构。这些结构包括 case 标号、goto case 语句、enum 成员声明、属性及其它常量声明。

如 7.15 节中所述, 一个 constant-expression 就是在编译期可被完全求值的一个表达式。由于产生 reference-type 的非 null 值的唯一方法是用 new 操作符, 且 new 操作符不允许出现在 constant-expression 中, reference-types 的常量的唯一可能值就是 null。

如果一个常量值的符号名合理, 但其类型不允许出现在常量声明中或其值不能在编译期被 constant-expression 计算, 那么, 可用 readonly 域来代替。

声明多个常量的一个常量声明就相当于具有相同属性修改函数及类型的单个常量的多次声明。例如:

```

class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}

```

就相当于：

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

常量可依赖于同一程序内的其它常量，只要这种依赖不是一个循环。编译函数以正确的顺序自动对这些常量声明求值。在下例中：

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}
```

编译函数先对 Y 求值，再对 Z 求值，最后对 X 求值，产生的结果为 10, 11 和 12。常量声明可以依赖其它程序中的常量，但这种依赖只在一个方向上有可能。参考上例，如果 A 和 B 分别在独立的程序中被声明， $A \cdot X$ 可能依赖 $B \cdot Z$ ，但 $B \cdot Z$ 不能同时依赖 $A \cdot Y$ 。

10.4 域 (Fields)

一个 field 就是一个成员，这个成员表示一个与某一类或对象有关的变量。field-declaration 引入一给定类型的一或多个域。

```
field-declaration:
    attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:
    field-modifier
    field-modifiers field-modifier

field-modifier:
    new
    public
    protected
    internal
    private
    static
    readonly
```

```

variable-declarators:
    variable-declarator
    variable-declarators , variable-declarator

variable-declarator:
    identifier
    identifier=variable-initializer

variable-initializer:
    expression
    array-initializer

```

一个 **field-declaration** 可能包括一系列 **attributes**、一个 **new** 修改函数、四个访问修改函数的一个有效组合、一个 **static** 修改函数及一个 **readonly** 修改函数。属性和修改函数适用于被 **field-declaration** 声明的所有成员。

field-declaration 的类型规定由这个声明引入的成员的类型的类型。此类型后是一个 **variable-declarators** 列表，每一个 **variable-declarator** 引入一个新成员。一个 **variable-declarator** 包括一个作为这个成员名字的 **identifier**，其后可选择地跟一个符号“=”及给定这个成员初始值的 **variable-initializer**。

一个域的 **type** 的可访问性必须至少写这个域本身相同。

在一个表达式，用一个 **simple-name** 或 **member-access** 的作用是得到一个域的值。用一个 **assignment** 的作用是修改一个域的值。而后缀增量和减量操作符及前缀增量和减量操作符既可得到一个域的值，也可修改这个值。

声明多个域的一个域声明过程相当于具有相同属性、修改函数及类型的单个域的多次声明。例如：

```

class A
{
    public static int X = 1, Y, Z = 100;
}

```

与下例一致

```

class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}

```

10.4.1 静态和实例域 (Static And Instance Fields)

当 **field-declaration** 含有 **static** 修改函数时，由这个声明所引入的域就是 **static fields**。当不存在 **static** 修改函数时，由这个声明所引入的域就是 **instance fields**。静态域和实例域是 C# 所支持的几种变量（参见第 5 章）中的两种，且有时被用作 **static variables** 和 **instance variables**。

静态域只标识存储空间。一个类不管产生多少个实例，只有一个静态域的拷贝。当声明一个静态域的类型登录时，这个静态域就开始存在，当声明它的类型退出时，这个静态域便停止存在。

一个类的每一个实例都包括这个类的所有实例域的一个独立的拷贝。当一个实例域类产生一个新的实例时，这个实例域就开始存在，当没有对那个实例的引用且这个实例的析构函数已执行时，这个实例域就停止存在。当一个域在式子 $e \cdot m$ 的 member-access（参见 7.5.4 节）中被引用时，如果 m 是一个静态域，则 e 必须表示一个类型；如果 m 是一个实例域， e 必须表示一个实例。静态和实例成员的不同点在 10.2.5 节中有进一步探讨。

10.4.2 readonly 域 (Readonly Fields)

当 field-declaration 含有 readonly 修改函数时，对由这个声明所引入的域的赋值只能作为这个声明的一部分而发生或发生在同一个类的构造函数内。但对域 readonly 的赋值只在下列上下文中被允许：

- 引入这个域（通过声明中的 variable-initializer）的 variable-declarator 中。
- 对于一个实例域来说，在包含这个域声明的类的实例构造函数中，或者，对于一个静态来说，在包含这个域声明的类的静态构造函数中。把 readonly 域作为 out 或 ref 参数来传递也只有在这些上下文中才有效。

在任何其它上下文中，试图对 readonly 域进行赋值或把它作为一个 out 或 ref 参数来传递都是错误的。

10.4.2.1 对常量使用静态只读域 (Using Static Readonly Fields For Constants)

当一个常量值的符号名正确，但其值的类型在 const 声明中不被允许或者这个值不能在编译期被计算时，static readonly 域是有用的。在下例中：

```
public class color
{
    public static readonly color black = new color(0, 0, 0);
    public static readonly color white = new color(255, 255, 255);
    public static readonly color red = new color(255, 0, 0);
    public static readonly color green = new color(0, 255, 0);
    public static readonly color blue = new color(0, 0, 255);

    private byte red, green, blue;
    public color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

成员 black、white、red、green 和 blue 不能被声明为 const 的成员，因为它们的值不能在

编译时被计算。然而，把这些成员声明为 `static readonly` 域却能得到上述效果。

10.4.2.2 常量和静态只读域翻译 (Versioning Of Constants And Static Readonly Fields)

常量和 `readonly` 域具有不同的二元翻译语义。当一个表达式引用一个常量时，这个常量的值在编译期被得到。但是，当一个表达式引用一个 `readonly` 域时，这个域的值直到运行期才可得到。下面是含有两个独立程序的一个应用：

```
namespace program1
{
    public class utils
    {
        public static readonly int x = 1;
    }
}

namespace program2
{
    class test
    {
        static void main() {
            console.WriteLine(program1.utils.x);
        }
    }
}
```

`program1` 和 `program2` 表示两个独立的编译程序。由于 `program1` `utils` `x` 被声明为一个静态的 `readonly` 域，由语句 `console.WriteLine` 输出的值在编译期是未知的，直到运行期才可获得。因此，如果 `x` 的值被改变，`program1` 被重新编译，则即使 `program2` 未被编译，语句 `console.WriteLine` 也将输出这个新值。然而，如果 `x` 是一个常量，那么，`x` 的值在编译 `program2` 时就应已被获得，且不受 `program1` 中变化的影响，直到 `program2` 被重新编译。

10.4.3 域初始化 (Field Initialization)

一个域的初始值就是这个域的类型默认值（参见 5.2 节）。当一个类登录时，所有静态域都被初始化为它们的默认值，而当一个类的实例产生时，所有的实例域都被初始化为它们的默认值。在这种默认初始化发生之前不可能看到一个域的值，因此，一个域不可能“未初始化”。下面的例子：

```
class test
{
    static bool b;
    int i;
```

```

static void main() {
    test t = new test();
    console.WriteLine("b = {0}, i = {1}", b, t.i);
}
}

```

输出:

```
b=false, i=0
```

因为当这个类登录时, `b` 被自动初始化为它的默认值, 而当这个类的一个实例产生时, `i` 被自动初始化为它的默认值。

10.4.4 变量初始化 (Variable Initializers)

域声明可能包括 `variable-initializers`。对于静态域来说, 变量初始化与类登录时所执行的赋值语句一致。对于实例域来说, 变量初始化与类的实例产生时所执行的赋值语句一致。下面的例子:

```

class test
{
    static double x = math.sqrt(2.0);
    int i = 100;
    string s = "hello";

    static void main() {
        test a = new test();
        console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}

```

输出:

```
x=1.4144213562373095,i=100,s=hello
```

因为对 `x` 的赋值发生在类登录时, 对 `i` 和 `s` 的赋值发生在此类的一个新的实例产生时,

对于所有的域, 包括具有变量初始化函数的域来说, 都首先被初始化为 10.4.3 节中所述的默认值, 然后, 再以原文顺序执行静态域初始化函数。同样, 当一个类的实例产生时, 所有的实例域都首先被初始化为它们的默认值, 然后再以原文顺序执行实例域初始化函数。

带有变量初始化函数的静态域有可能以其默认值状态出现, 尽管这种情况被强烈排斥。下面的例子就是这种情况:

```

class test
{
    static int a = b + 1;
}

```

```

static int b = a + 1;
static void main() {
    console.WriteLine("a = {0}, b = {1}", a, b);
}
}

```

尽管 `a` 和 `b` 被循环定义，但这个程序是合法的。它输出：

```
a=1, b=2
```

因为静态域 `a` 和 `b` 在它们的初始化函数被执行之前已被初始化为 0（`int` 的默认值）。当 `a` 的初始化函数运行时，`b` 的值是 0，因此，`a` 被初始化为 1。当 `b` 的初始化函数运行时，`a` 的值已经是 1，因此，`b` 被初始化为 2。

10.4.4.1 静态域初始化 (Static Field Initialization)

一个类的静态域变量初始化函数与一系列赋值一致，这些赋值一进入这个类的构造函数就立即被执行。变量初始化函数以它们在这个类的声明过程中出现的原文顺序被执行。类的登录及初始化过程将在 10.11 节中进一步描述。

10.4.4.2 实例域初始化 (Instance Field Initialization)

一个类的实例域变量初始化函数同一系列赋值一致，这些赋值一进入这个类的实例构造函数就立即被执行。变量初始化函数以它们在这个类的声明过程中出现的原文顺序被执行。类实例的产生及初始化过程将在 10.10 节中进一步描述。

实例域的变量初始化函数不能引用正在产生的实例。因此，在变量初始化函数中引用 `this` 是错误的，同样，变量初始化函数通过 `simple-name` 引用实例成员也是错误的。在下例中：

```

class a
{
    int x = 1;
    int y = x + 1;    // error, reference to instance member of this
}

```

`y` 的变量初始化函数是错误的，因为它引用了一个正在被产生的实例的成员。

10.5 方法 (Methods)

一个 `method` 就是一个成员，这个成员执行可由对象或类执行的计算或其它功能。

方法用 `method-declarations` 来声明：

```

method-declaration:
method-header method-body
method-header:
attributesopt method-modifiersopt return-type member-name
( formal-parameter-listopt )

```

```

method-modifiers:
method-modifier
method-modifiers  method-modifier
method-modifier:
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern
return-type:
type
void
member-name:
identifier
interface-type  .  identifier
method-body:
block
;

```

一个 `method-declaration` 可能包括一系列 `attributes`，一个 `new` 修改函数，一个 `extern` 修改函数，四个访问修改函数的有效组合及修改函数 `static`、`virtual`、`override` 和 `abstract` 的有效组合。另外，含有修改函数 `override` 的方法也可以含有修改函数 `sealed`。

除一种情况外，修改函数 `static`、`virtual`、`override` 和 `abstract` 互相排斥。修改函数 `abstract` 和 `override` 可一起使用，以便抽象方法可重载一个虚拟方法。方法声明的 `return-type` 指定由这个方法计算和返回的值类型。如果这个方法不返回一个值，则其 `return-type` 就是 `void`。

`member-name` 指定方法的名字。除非方法是一个显式接口成员执行，否则，这个 `member-name` 只是一个 `identifier`。对于显式接口成员执行来说，`member-name` 包括一个 `interface-type`，然后是一个 “.” 和 `identifier`。

可选择的 `formal-parameter-list` 说明了这个方法的参数（参见 10.5.1 节）。

`return-type` 和在一个方法的 `formal-parameter-list` 中被引用的每一个类型的可访问性都必须至少与这个方法本身相同。对于方法 `abstract` 和 `extern` 来说，`method-body` 只包括一个分号。对于所有其它方法来说，`method-body` 包括一个 `block`，当方法被引用时，`block` 就规定语句的执行。方法的名字及形参列表就定义其签名。但方法的签名包括其名字及数量、修改函数及形参的类型。返回类型并不是方法签名的一部分，也不是其形参的名字。方法的名字必须

与在同一类中声明的所有其它非方法的名字不同。另外，方法的签名也必须与在同一类中声明的所有其它方法的签名不同。

10.5.1 方法参数 (Method Parameters)

方法的参数由其 formal-parameter-list 声明。

```
formal-parameter-list:
    fixed-parameters
    fixed-parameters , parameter-array
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters , fixed-parameter

fixed-parameter:
    attributesopt parameter-modifieropt type identifier

parameter-modifier:
    ref
    out

parameter-array:
    attributesopt params array-type identifier
```

形参列表包括一或多个 fixed-parameters，后面是一个可选的 parameter-array，这些部分之间都由逗号隔开。

一个 fixed-parameter 包括一个可选的集合，这个集合包括 attributes、可选的 ref 或 out 修改函数、type 及 identifier。每一个 fixed-parameter 都声明一个给定名字类型的参数。

一个 parameter-array 包括一个可选的集合，这个集合包括 attributes、params 修改函数、array-type 及 identifier。一个参数数组声明一个给定名字数组类型的单一参数。一个参数数组的 array-type 必须是一维数（参见 12.1 节）。在方法引用过程中，参数数组允许指定给定数组类型的一个自变量或这个数组元素类型的零或多个自变量。关于参数数组将在 10.5.1.4 节中进一步描述。

方法声明为参数和局部变量产生一个独立的声明空间。名字是通过这个方法的形参列表及其块中的局部变量声明而引入的。这个方法声明空间的所有名字都必须是唯一的。因此，如果一个参数或局部变量与另一参数或局部变量的名字相同，则错误发生。

方法引用产生其形参和局部变量的一个拷贝（只对这个引用而言）。且这个引用的自变量列表赋给新产生的形参值或变量引用。在一个方法的 block 内，形参在 simple-name 表达式中可被其标识符引用。

形参有四种：

- 值参数，声明时不需要任何修改函数。
- 引用参数，声明时带有 ref 修改函数。
- 输出参数，声明时带有 out 修改函数。
- 参数数组，声明时带有 params 修改函数。

如 3.6 节中所述, 修改函数 `ref` 和 `out` 是方法签名的一部分, 但修改函数 `params` 不是。

10.5.1.1 值参数 (Value Parameters)

声明时不带修改函数的参数就是一个值参数。值参数与局部变量一致, 局部变量的初始值是从由这个方法引用提供的相应的自变量中得到的。当某一形参是一个值参数时, 在方法引用中的相应的自变量必须是一类型的表达式, 这个类型通过隐式转换可转化为该形参类型。

允许方法给一个值参数赋新值。这样的赋值只影响由值参数所表示的局部存储空间——不影响方法引用中所给定的实际自变量。

10.5.1.2 引用参数 (Reference Parameters)

声明时带有修改函数 `ref` 的参数就是一个引用参数。与值参数不同, 引用参数不产生新的存储空间。而是与给定在该方法引用中作为自变量的变量表示相同的存储空间。

当某一形参是一个引用参数时, 方法引用中相应的自变量必须包括关键词 `ref`, 其后是与这个形参类型相同的一个 `variable-reference`。变量在其被作为引用参数传递之前必须被明确赋值。在方法内, 引用参数总被认为是明确赋值的。

下例:

```
class test
{
    static void swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void main() {
        int i = 1, j = 2;
        swap(ref i, ref j);
        console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

输出:

```
i=2, j=1
```

对于 `main` 中的 `swap` 的引用来说, `x` 表示 `i`, `y` 表示 `j`。因此, 这个引用具有交换 `i` 和 `j` 值的效果。

在使用引用参数的方法中, 可用多个名字表示相同的存储空间。在下例中:

```
class a
{
    string s;
    void f(ref string a, ref string b) {
        s = "one";
    }
}
```

```
a = "two";
b = "three";
}
void g() {
    f(ref s, ref s);
}
}
```

g 中 f 的引用过程把对 s 的引用既传递给 a，又传递给 b。因此，对于该引用来说，名字 s、a 和 b 都指的是同一个存储空间，这三个赋值都修改实例域 s。

10.5.1.3 输出参数 (Output Parameters)

声明时带有修改函数 out 的参数就是一个输出参数。与引用参数相同，输出参数不产生新的存储空间。而是与给定在这个方法引用中作为自变量的变量具有相同的存储空间。

当形参是一个输出参数时，在方法引用中相应的自变量必须包括关键词 out，其后是一个与这个形参类型相同的 variable-reference。变量不需明确赋值即可作为输出参数传递，但变量作为形参引用之后就被认为是经过明确赋值的。在一个方法内，同一个局部变量相同，一个输出参数最初被认为是未被赋值的且在其值被应用之前必须被明确赋值。方法的每一个输出参数在该方法返回之前必须被明确赋值。输出参数一般用于产生多个返回值的方法中。例如：

```
class test
{
    static void splitpath(string path, out string dir, out string name) {
        int i = path.length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\ ' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.substring(0, i);
        name = path.substring(i);
    }

    static void main() {
        string dir, name;
        splitpath("c:\\windows\\system\\hello.txt", out dir, out name);
        console.WriteLine(dir);
        console.WriteLine(name);
    }
}
```


这个例子输出:

```
c: \windows\system\hello*.*
```

注意: 变量 `dir` 和 `name` 在其被作为 `splitpath` 传递之前可以是未被赋值的, 且在调用之后, 就认为被明确赋值。

10.5.1.4 参数数组 (Parameter Arrays)

声明时带有修改函数 `params` 的参数就是一个参数数组。如果一个形参列表包括一个参数数组, 则它必须是这个列表中最后的参数且必须是单维数组类型。例如, 类型 `string[]` 和 `string[]` 可被用作一个形参数组的类型, 但类型 `string[,]` 不能。不能把修改函数 `params` 和 `ref` 及 `out` 组合在一块儿。

参数数组允许自变量在方法引用中以下列两种方式之一被指定:

- 一个参数数组的给定自变量可以是一类型的单个表达式, 这个类型可被隐式转化为这个参数数组类型。在这种情况下, 这个参数数组就相当于一个值参数。
- 在另一种方式中, 引用可以为这个参数数组指定零或多个自变量, 这里, 每一个自变量都是一类型的表达式, 这个类型可被隐式转化为该参数数组的元素类型。在这种情况下, 这个引用产生该参数数组类型的一个实例, 其长度与自变量数一致, 这个引用还用给定的自变量值初始化该数组实例的元素, 并把新产生的数组实例作为实际自变量。

在引用中, 除自变量数目允许变化外, 参数数组就相当于同一类型的一个值参数。下面的例子:

```
class test
{
    static void f(params int[] args) {
        console.WriteLine("array contains {0} elements:", args.Length);
        foreach (int i in args) console.Write(" {0}", i);
        console.WriteLine();
    }

    static void main() {
        int[] a = {1, 2, 3};
        f(a);
        f(10, 20, 30, 40);
        f();
    }
}
```

输出:

```
array contains 3 elements: 1 2 3
array contains 4 elements: 10 20 30 40
array contains 0 elements:
```

f 的第一次引用只把数组 a 当作一个值参数来传递。f 的第二次引用就用给定的元素值自动产生一个四元素的 int[], 并把这个数组实例作为一个值参数来传递。同样, f 的第三次引用产生一个零元素的 int[], 并把那个实例作为一个值参数来传递。第二次和第三次引用都可书写为:

```
    f(new int[] {10, 20, 30, 40});
```

```
    f (new int[] {});
```

在执行重载分解时, 带有一个参数数组的方法既可以其正常形式被应用, 也可以其扩展形式被应用。只有当一个方法的正常形式不可利用或与这个扩展形式具有相同签名的方法还未在同一类型中被声明时, 才可利用其扩展形式。

下面的例子:

```
class test
{
    static void f(params object[] a) {
        console.WriteLine("f(object[])");
    }

    static void f() {
        console.WriteLine("f()");
    }

    static void f(object a0, object a1) {
        console.WriteLine("f(object,object)");
    }

    static void main() {
        f();
        f(1);
        f(1, 2);
        f(1, 2, 3);
        f(1, 2, 3, 4);
    }
}
```

输出:

```
f();
f(object[]);
f(object,object);
f(object[]);
f(object[]);
```

在这个例子中, 带有一个参数数组的这个方法的两种可能的扩展形式都已作为常规方法

出现在这个类内。因此，当执行重载分解时不考虑这些扩展形式，因此，第一和第三个方法引用选择常规方法。当一个类声明一个带有参数数组的方法时，有些扩展形式也包括在这个类内，这并不奇怪。这样做可避免带有参数数组方法的扩展形式被引用时所发生的数组实例分配。

当一个参数数组的类型是 `object[]` 时，在方法的标准形式和一个 `object` 参数的扩展形式之间就会存在一种潜在的混乱。混乱的原因就是 `object[]` 本身可被隐式转化为类型 `object`。然而，混乱并不产生任何问题，因为，如果需要的话，可通过插入一个 `cast` 来解决。

下例：

```
class test
{
    static void f(params object[] args) {
        foreach (object o in a) {
            console.write(o.gettype().fullname);
            console.write(" ");
        }
        console.writeline();
    }

    static void main() {
        object[] a = {1, "hello", 123.456};
        object o = a;
        f(a);
        f((object)a);
        f(o);
        f((object[])o);
    }
}
```

输出：

```
system.int32 system.string system.double
system.object[]
system.object[]
system.int32 system.string system.double
```

在 `f` 的第一个和最后一个引用中，其正常形式是可利用的，因为从其自变量类型到参数类型（都是类型 `object[]`）存在一个隐式转换。因此，重载分解选择 `f` 的标准形式，且这个自变量作为一个常规的值参数被传递。

在第二和第三个引用中，`f` 的标准形式是不可用的，因为从其自变量到参数类型之间不存在隐式转换（类型 `object` 不能被隐式转化为类型 `object[]`）。然而，`f` 的扩展形式是可用的，因此，它被重载分解选择。结果产生一个一元素的 `object[]`，而该数组的单个元素被初始化为给定的自变量值（这个值本身就是对一个 `object[]` 的引用）。

10.5.2 静态和实例方法 (Static And Instance Methods)

如果一个方法声明含有修改函数 `static`, 则这个方法就是一个静态方法。如果没有修改函数 `static`, 这个方法就是一个实例方法。

静态方法对一特定的实例不起作用, 且在静态方法中引用 `this` 是错误的。而且, 静态方法中含有修改函数 `virtual`、`abstract` 或 `override` 也是错误的。

实例方法对一个类的给定实例起作用, 且这个实例可作为 `this` 被访问。

静态及实例成员的不同点在 10.2.5 节中有进一步探讨。

10.5.3 虚拟方法 (Virtual Methods)

如果一个实例方法声明含有修改函数 `virtual`, 则这个方法就是一个非虚拟方法。

含有修改函数 `virtual` 的方法声明不能含有下列任何一个修改函数: `static`、`abstract` 或 `override`, 否则, 就是错误的。

非虚拟方法的执行是恒定的, 不管这个方法是在声明它的类的一个实例中被引用还是在派生类的实例中被引用这个执行总是相同的。而虚拟方法的执行则可能被派生类改变。改变一个继承虚拟方法执行的过程称为 `overriding` 这个方法 (参见 10.5.4 节)。

在虚拟方法引用过程中, 发生引用的 `non-time type` 决定要引用的实际方法执行。在非虚拟方法引用中, 实例的 `compile-time type` 就是决定因素。在合适条件下, 当一个名字为 `n` 且带有一个自变量列表 `a` 的方法在编译期类型为 `c`, 运行期类型为 `r` (这里, `r` 是 `c` 或者由 `c` 派生的一个类) 的一个实例中被引用时, 其过程如下:

- 首先, 对 `c.n` 和 `a` 进行重载分解以便从在 `c` 中声明并被 `c` 继承的方法的集合中选择一特定的方法。这在 7.5.5.1 节中有描述。
- 然后, 如果 `m` 是一个非虚拟方法, 则引用 `m`。
- 否则, 如果 `m` 是一个非虚拟方法, 则引用与 `r` 有关的 `m` 的最原始的执行。

对于每一个在类中声明或被这个类继承的方法来说, 都存在一个与那个类有关的方法的 `most derived implementation`。与一个类 `r` 有关的虚拟方法 `m` 的最原始的执行的确定过程如下:

- 如果 `r` 包括 `m` 的正在输入的 `virtual` 声明, 则它就是 `m` 最原始的执行。
- 否则, 如果 `r` 包括 `m` 的一个 `override`, 则它就是 `m` 最原始的执行。
- 否则, `m` 的最原始的执行就与 `r` 的直属基类的最原始执行相同。

下面的例子说明虚拟与非虚拟方法之间的不同点。

```
class a
{
    public void f() { console.writeline("a.f"); }

    public virtual void g() { console.writeline("a.g"); }
}

class b: a
{
    new public void f() { console.writeline("b.f"); }
```

```

        public override void g() { console.WriteLine("b.g"); }
    }

    class test
    {
        static void main() {
            b b = new b();
            a a = b;
            a.f();
            b.f();
            a.g();
            b.g();
        }
    }

```

在这个例子中，a 引入一个非虚拟方法 f 及一个虚拟方法 g。类 b 引入一个 new 非虚拟方法 f，因此也就隐藏了被继承的 f，并且也 overrides 被继承的方法 g。这个例子输出：

```

a.f
b.f
b.g
b.g

```

注意：语句 a.g() 引用的是 b.g 而不是 a.g。这是因为，这个实例是运行期类型（即 b），而不是它的编译期类型（即 a）决定要引用的实际方法执行。

由于方法允许隐藏被继承的方法，因此，一个类含有几个具有相同签名的方法是可能的。这并不产生混淆问题，因为除最原始的方法外都被隐藏了。在下例中：

```

class a
{
    public virtual void f() { console.WriteLine("a.f"); }
}

class b: a
{
    public override void f() { console.WriteLine("b.f"); }
}

class c: b
{
    new public virtual void f() { console.WriteLine("c.f"); }
}

class d: c
{

```

```
public override void f() { console.WriteLine("d.f"); }  
}  
class test  
{  
    static void main() {  
        d d = new d();  
        a a = d;  
        b b = d;  
        c c = d;  
        a.f();  
        b.f();  
        c.f();  
        d.f();  
    }  
}
```

类 c 和 d 含有两个具有相同签名的虚拟方法：一个是由 a 引入的，另一个是由 c 引入的，由 c 引入的方法隐藏由 a 引入的方法。因此，d 中的重载声明重载由 c 引入的方法，且 d 不可能重载由 a 引入的方法。这个例子输出：

```
bf  
bf  
df  
df
```

注意：通过从不隐藏方法的一个派生较小的类型访问 d 的实例可引用被隐藏的虚拟方法。

10.5.4 重载方法 (Override Methods)

当一个实例方法声明过程含有修改函数 `override` 时，这个方法就是一个 `override method`。

重载方法重载一个具有相同签名的继承虚拟方法。而且，虚拟方法声明 `introduces` 的一个新方法，重载方法声明通过提供该方法的一个新的执行 `specializes` 一个正在被继承的虚拟方法。

重载方法声明不能含有下列修改函数：`new static` 或 `virtual`，否则，就是错误的。重载方法声明可以含有修改函数 `abstract`。它使得虚拟方法被抽象方法重载。

被 `override` 声明重载的方法称为 `overridden base method`。对于在类 c 中声明的一个重载方法 m 来说，被重载的基本方法取决于对 c 的每一个基类的检查，这个检查从 c 的直属基类开始，然后是每一个连续的直属基类，直至找到一个与 m 具有相同签名的可访问的方法。为了寻找被重载的基本方法，如果一个方法是 `public`、`protected`、`protected internal` 或 `internal` 且是在与 c 同样的程序中声明的，那么，这个方法就是可访问的。

对于一个重载声明来说，除非下列条件都符合，否则，就会出现编译期错误：

- 按照上面的方法，可以找到一个被重载的基本方法。
- 被重载的基本方法是一个虚拟的，抽象的或重载的方法。也就是说，被重载的基本方法不是静态的或非虚拟的。
- 被重载的基本方法不是一个封装方法。
- 重载声明与被重载的基本方法具有相同声明的可访问性。也就是说，重载声明不能改变虚拟方法的可访问性。

重载声明用 `base-access`（参见 7.5.8 节）可访问被重载的基本方法。在下例中：

```
class a
{
    int x;

    public virtual void printfields() {
        console.WriteLine("x = {0}", x);
    }
}

class b: a
{
    int y;

    public override void printfields() {
        base.printfields();
        console.WriteLine("y = {0}", y);
    }
}
```

`b` 中的引用过程 `base printfields()` 引用在 `a` 中声明的方法 `printfields`，一个 `base-access` 使得虚拟引用方法无效，而只是把这个基本方法当作一个非虚拟方法来对待。如果 `b` 中的引用书写为 `((a) this) print fields()`，那么，它将循环引用在 `b` 中声明的方法 `print fields`。

只有当一个方法含有修改函数 `override` 时，它才能重载另一个方法。在所有其它情况下，一个与某将要被继承的方法具有相同签名的方法只隐藏这个被继承的方法。在下例中 `b` 中的方法下不包括修改函数 `override`，因此也就不重载 `a` 中的方法 `f`：

```
class a
{
    public virtual void f() {}
}

class b: a
{
    public virtual void f() {}    // warning, hiding inherited f()
}
```

相反, b 中的方法 f 隐藏 a 中的这个方法, 且有警告出现, 因为这个声明不包括修改函数 new 。

在下例中 b 中的方法 f 隐藏从 a 继承的虚拟的方法 f:

```
class a
{
    public virtual void f() {}
}
class b: a
{
    new private void f() {}           // hides a.f within b
}
class c: b
{
    public override void f() {} // ok, overrides a.f
}
```

由于 b 中新的 f 具有局部可访问性, 因此, 它的范围只包括 b 的类主体而不延伸至 c。因此, 允许 c 中 f 的声明重载从 a 继承的 f。

10.5.5 封装方法 (Sealed Methods)

当一个实例方法声明含有修改函数 sealed 时, 这个方法就是一个 sealed method。封装方法重载一个具有相同签名的继承虚拟方法。虚拟方法声明引入一个新的方法, 重载方法声明则通过提供这个方法新的执行特化一个现存的继承虚拟方法。

一个重载方法也可被修改函数 sealed 标记。这个修改函数的作用是防止派生的类进一步重载这个方法。修改函数 sealed 只能与修改函数 override 联合使用。在下例中:

```
class a
{
    public virtual void f() {
        console.WriteLine("a.f");
    }
    public virtual void g() {
        console.WriteLine("a.g");
    }
}
class b: a
{
    sealed override public void f() {
        console.WriteLine("b.f");
    }
}
```



```

override public void g() {
    console.WriteLine("b.g");
}
}
class c: b
{
    override public void g() {
        console.WriteLine("c.g");
    }
}

```

类 `b` 提供了两个重载方法：具有修改函数 `sealed` 的方法 `f` 和没有此修改函数的方法 `g`。`b` 中 `modifier` 的作用是阻止 `c` 进一步重载 `f`。

10.5.6 抽象方法 (Abstract Methods)

当一个实例方法声明含有修改函数 `abstract` 时，这个方法就是一个 `abstract method`。抽象方法也是一个默认的虚拟方法。

抽象方法声明引入一个新的虚拟方法，但不提供这个方法的一个执行，而非抽象的派生类则需通过重载该方法提供它们自己的执行。由于抽象方法不提供实际的执行，因此，抽象方法的 `method-body` 只包括一个分号。

抽象方法声明只允许出现在抽象类（参见 10.1.1.1 节）中。

抽象方法声明不能含有修改函数 `static` 或 `virtual`，否则就是错误的。在下例中类 `shape` 定义一个几何形对象的抽象符号，这个对象可以修饰它自己：

```

public abstract class shape
{
    public abstract void paint(graphics g, rectangle r);
}
public class ellipse: shape
{
    public override void paint(graphics g, rectangle r) {
        g.drawellipse(r);
    }
}
public class box: shape
{
    public override void paint(graphics g, rectangle r) {
        g.drawrect(r);
    }
}

```

方法 `paint` 是抽象的，因为它没有具体的默认执行。类 `ellipse` 和 `box` 都是具体的 `shape` 执行。因为这些类都是非抽象的，它们需要重载方法 `paint` 并提供一个实际的执行。

`base-access`（参见 7.5.8 节）不能引用抽象方法，否则，就是错误的。在下例中在引用过程 `base.f()` 处显示一个错误，因为它引用了一个抽象方法：

```
class a
{
    public abstract void f();
}

class b: a
{
    public override void f() {
        base.f();                // error, base.f is abstract
    }
}
```

抽象方法声明可以重载虚拟方法。在派生类中，这使得抽象类对方法进行强行重新执行，且使得这个方法原来的执行不可再利用。在下例中类 `a` 声明一个虚拟方法，类 `b` 用一个抽象方法重载这个方法，类 `c` 重载以提供它自己的执行：

```
class a
{
    public virtual void f() {
        console.WriteLine("a.f");
    }
}

abstract class b: a
{
    public abstract override void f();
}

class c: b
{
    public override void f() {
        console.WriteLine("c.f");
    }
}
```

10.5.7 外部方法 (External Methods)

当一个方法声明含有修改函数 `extern` 时，这个方法就是一个 `external method`。外部方法用 C# 以外的语言执行。由于外部方法声明不提供实际执行，因此，外部方法的 `method-body` 只包括一个分号。

修改函数 `extern` 的典型用法是与一个属性 `dllimport` (参见 13.8 节) 相结合, 允许外部方法由 DLLs (动态链接库) 执行。执行环境可支持能够提供外部方法执行的途径。

外部方法声明不含有修改函数 `abstract`, 否则, 就是错误的。当一外部方法包括一个 `dllimport` 属性时, 该方法声明也必须包括一个修饰函数 `static`。

下面的例子说明了修饰函数 `extern` 和属性 `dllimport` 的用法:

```
class path
{
    [dllimport("kernel32", setlasterror=true)]
    static extern bool createdirectory(string name, securityattributes sa);

    [dllimport("kernel32", setlasterror=true)]
    static extern bool removedirectory(string name);

    [dllimport("kernel32", setlasterror=true)]
    static extern int getcurrentdirectory(int bufsize, stringbuilder buf);

    [dllimport("kernel32", setlasterror=true)]
    static extern bool setcurrentdirectory(string name);
}
```

10.5.8 方法主体 (Method Body)

一个方法声明的 `method-body` 包括一个 `block` 或一个分号。

抽象及外部方法声明不提供具体的方法执行过程, 且抽象或外部方法的方法主体只包括一个分号。对所有其它方法来说, 方法主体是一个包括引用这个方法时所执行的语句的块 (参见 8.2 节)。

当一个方法的返回类型是 `void` 时, 这个方法主体中的 `return` 语句 (参见 8.9.4 节) 不允许指定一个表达式。如果一个无效方法的方法主体正常地完成其执行 (也就是说, 如果控制到达方法主体的结束), 则这个方法只返回到调用者。

当一个方法的返回类型不是 `void` 时, 方法主体中的每一个 `return` 语句都必须指定可隐式转化为返回类型的一个类型的表达式。一个值返回方法的方法主体的执行需要在一指定表达式的 `return` 语句域抛出一个异常的 `throw` 语句终止。如果这个方法主体的执行可正常完成, 那么, 就是错误的。也就是说, 在一个值返回方法中, 不允许控制到达这个方法主体的结束。

在下例中值返回方法 `f` 是错误的, 因为控制可到达这个方法主体的终点:

```
class a
{
    public int f() {}           // error, return value required

    public int g() {
        return 1;
    }
}
```

```
public int h(bool b) {  
    if (b) {  
        return 1;  
    }  
    else {  
        return 0;  
    }  
}  
}
```

g 和 h 方法是正确的，因为所有可能的执行路径都在指定一返回值的返回语句终止。

10.5.9 方法重载 (Method Overloading)

方法重载分解规则如 7.4.2 节中所述。

10.6 属性 (Properties)

一个 property 就是一个成员，它提供对一个对象或类的访问。属性包括：字符串长度、汉字的大小、窗口的标题、惯用名等等。属性是域的自然延伸——都称为与类型有关的成员，且访问域和属性的句法也是相同的。然而，与域不同，属性不表示存储空间。而属性则具有指定要执行语句的 *accessors*，以便读或写它们的值。因此，属性为与对象属性的读和写有关的操作提供了一种方法，而且它们允许这样的属性被计算。

属性用 *property-declarations* 来声明：

```
property-declaration:  
    attributesopt    property-modifiersopt    type    member-name  
    { accessor-declarations }  
property-modifiers:  
    property-modifier  
property-modifiers    property-modifier  
property-modifier:  
    new  
    public  
    protected  
    internal  
    private  
    static  
    virtual  
    sealed  
    override  
    abstract
```

```

member-name:
    identifier
    interface-type . identifier

```

一个 `property-declaration` 可能包括一系列 `attributes`，一个 `new` 修改函数、四个可访问修改函数的有效组合、修改函数 `static`、`virtual`、`override` 及 `abstract` 的一个有效组合。另外，含有 `override` 修改函数的属性也可以含有 `sealed` 修改函数（参见 10.5.5 节）。

除一种情况外，修改函数 `static`、`virtual`、`override` 和 `abstract` 互相排斥。修改函数 `abstract` 和 `override` 可一块儿用，其作用是使抽象属性可以重载虚拟属性。

一个属性声明的 `type` 规定由这个声明引入的属性的类型，而 `member-name` 规定这个属性的名字。除非这个属性是一个显式接口成员执行，否则，这个 `member-name` 只是一个 `identifier`。对于一个显式接口成员执行（参见 13.4.1 节）来说，`member-name` 包括一个 `interface-type`，其后是一个 “.” 及 `identifier`。

一个属性的 `type` 的可访问性必须至少与这个属性本身相同（参见 3.5.4 节）。

必须包括在符号 “{” 和 “}” 之内的 `accessor-declarations` 声明这个属性的可访问函数（参见 10.6.2 节）。这个可访问函数指定与读和写这个属性有关的可执行语句。即使访问一个属性的语句与访问域的语句相同，也不能把属性看作是一个变量。因此，不可能把一个属性当作 `ref` 或 `out` 参数来传递。

10.6.1 静态属性 (Static Properties)

当一个属性的声明过程含有 `static` 修改函数时，这个属性就是一个 `static property`。当不存在 `static` 修改函数时，这个属性就是一个 `instance property`。

静态属性与特定的实例无关，且在静态属性的可访问函数内引用 `this` 是错误的。而且，在静态属性中含有 `virtual` `abstract` 或 `override` 修改函数也是错误的。

实例属性与一个类的某一给定实例有关，且在这个属性的访问函数内，这个实例可作为 `this` 被访问（参见 7.5.7 节）。

当一个属性在式子 `e.m` 的 `member-access`（参见 7.5.4 节）中被引用时，如果 `m` 是一个静态属性，`e` 必须表示一个类型；如果 `m` 是一个实例属性，则 `e` 必须表示一个实例。

静态与实例成员的不同点在 10.5 节中有进一步探讨。

10.6.2 访问函数 (Accessors)

一个属性的 `accessor-declarations` 指定与读和写这个属性有关的可执行语句。

```

accessor-declarations:
    get-accessor-declaration    set-accessor-declarationopt
    set-accessor-declaration    get-accessor-declarationopt
    get-accessor-declaration:
        attributesopt get  accessor-body
    set-accessor-declaration:
        attributesopt set  accessor-body

```

```
accessor-body:  
block
```

访问函数的声明过程包括一个 `get-accessor-declaration` 或 `set-accessor-declaration` 或者两者都包括。每一个访问函数声明包括符号 `get` 或 `set`，其后是一个 `accessor-body`。对于 `abstract` 属性来说，每一个指定访问函数的 `accessor-body` 只是一个分号。对于所有其它访问函数来说，`accessor-body` 就是一个 `block`，这个 `block` 规定当这个访问函数被引用时要执行的语句。

`get` 访问函数与一个具有这个属性类型返回值的无参数方法一致。除作为赋值的目标外，当一属性在表达式中被引用时，这个属性的 `get` 访问函数就被引用以计算这个属性（参见 7.1.1 节）的值。`get` 访问函数的主体必须遵守 10.5.8 节中所述的值返回方法规则。尤其是，一个 `get` 访问函数主体中的所有 `return` 语句必须指定可隐式转化为这个属性类型的一个表达式。而且，要求 `get` 访问函数在遇到 `return` 语句或 `throw` 语句时终止执行，控制不能到达 `get` 访问函数主体的结束点。

`set` 访问函数与方法一致，这个方法带有属性类型的一个值参数及返回类型 `void`。`set` 访问函数的隐式参数总被命名为 `value`。当属性被作为一个赋值的目标而引用时，带有自变量的 `set` 访问函数就被引用，这个自变量提供一个新值（参见 7.13.1 节）。一个 `set` 访问函数的主体必须遵守 10.5.8 节中所述的 `void` 方法的规则。尤其是，这个 `set` 访问函数主体中的 `return` 语句不能指定一个表达式。由于 `set` 访问函数固定具有一个名字为 `value` 的参数，因此 `set` 访问函数中的局部变量声明不能再使用那个名字，否则，就是错误的。

根据 `get` 和 `set` 访问函数的存在与否，属性可分为以下几类：

- 既含有 `get` 访问函数，又含有 `set` 访问函数的属性为 `read-write` 属性。
- 只含有 `get` 访问函数的属性为 `read-only` 属性。`read-only` 属性不能是一赋值的对象。
- 只含有 `set` 访问函数的属性为 `write-only` 属性。除非作为一赋值的目标，否则，在表达式中不能引用 `write-only` 属性。

注意：在 Microsoft .net 运行期间，当一个类声明类型 `t` 的一个属性 `x` 时，它不能再声明具有下列签名的方法，否则就是错误的：

```
t get_x();  
void set_x(t value);
```

Microsoft .net 运行期为不支持属性的程序语言保留这些签名。注意，这并不意味着在 c# 程序中可以用方法语句访问属性或用属性语句访问方法，它的意思只是，具有这种模式的属性和方法不能同时出现在同一个类中。

在下例中：

```
public class button: control  
{  
    private string caption;  
    public string caption {  
        get {  
            return caption;  
        }  
    }  
}
```

```

    }
    set {
        if (caption != value) {
            caption = value;
            repaint();
        }
    }
}

public override void paint(Graphics g, Rectangle r) {
    // painting code goes here
}
}

```

button 控制声明一个公开的 caption 属性。这个 caption 属性的 get 访问函数返回一个存储在局部 caption 域中的字符串。如果新值与当前值不同，则 set 访问函数就要检查，如果情况确实如此，则它将存储新值并重新启动控制器。属性通常遵循上面所示的模式：get 访问函数只返回存储在局部域中的一个值，而 set 访问函数则修改这个局部域并执行使这个对象的状态完全更新所需的额外操作。

给定上述 button 类，下面是这个 caption 属性应用的一个例子：

```

button okbutton = new button();
okbutton.caption = "ok";           // invokes set accessor
string s = okbutton.caption;       // invokes get accessor

```

这里，set 访问函数的作用是把一个值赋给这个属性，get 访问函数的作用是在一个表达式中引用这个属性。

一个属性的 get 和 set 访问函数并非性质截然不同的成员，单独声明一个属性的这些访问函数是不可能的。下例并不声明一个读写属性。而是声明两个具有相同名字的属性，一个为只读，一个为只写：

```

class a
{
    private string name;
    public string name {           // error, duplicate member name
        get { return name; }
    }
    public string name {           // error, duplicate member name
        set { name = value; }
    }
}

```

由于在同一类中声明的两个成员不能具有相同的名字，因此，这个例子导致一个编译期错误发生。

当一个派生类声明一个与继承类型名字相同的属性时，这个派生属性就隐藏与读写都有关的继承属性。

下例中 `b` 中的 `p` 属性隐藏 `a` 中与读写都有关的 `p` 属性：

```
class a
{
    public int p {
        set {...}
    }
}

class b: a
{
    new public int p {
        get {...}
    }
}
```

因此，在下列语句中对 `b.p` 的赋值导致出现一个错误，因为 `b` 中的只读 `p` 属性隐藏 `a` 中的只写 `p` 属性：

```
b b = new b();
b.p = 1;           // error, b.p is read-only
((a)b).p = 1;      // ok, reference to a.p
```

但注意，`cast` 可访问这个被隐藏的 `p` 属性。与公共域不同，属性中对象的内部状态和其公共接口是互相独立的。

看下例：

```
class label
{
    private int x, y;
    private string caption;
    public label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }
    public int x {
        get { return x; }
    }
    public int y {
        get { return y; }
    }
    public point location {
```



```

        get { return new point(x, y); }
    }
    public string caption {
        get { return caption; }
    }
}

```

这里，label 类用两个 int 域（x 和 y）存储其地址。这个地址既作为 x 和 y 属性，又作为类型 point 的一个 location 属性。如果在 label 将来的版本中，把这个地址作为一个内部的 point 来存储更方便的话，则这个变化不会影响该类的公共接口：

```

class label
{
    private point location;
    private string caption;
    public label(int x, int y, string caption) {
        this.location = new point(x, y);
        this.caption = caption;
    }
    public int x {
        get { return location.x; }
    }
    public int y {
        get { return location.y; }
    }
    public point location {
        get { return location; }
    }
    public string caption {
        get { return caption; }
    }
}

```

如果 x 和 y 是 public readonly 域，则对于这个 label 类来说不可能发生这样的变化。

通过属性显示状态并不一定比直接显示域效率低。尤其是，当一个属性是非虚拟的且只包括数量很少的代码时，执行环境可能会用这个访问函数的实际代码代替对访问函数的调用。这个过程就是 inlining，它使得属性访问与域访问的效率相同，并保留属性已增强的灵活性。

由于从概念上来说，引用一个 get 访问函数就相当于读一个域的值，因此，如果 get 访问函数具有可见的副作用，就是一种不良的程序风格。在下例中 next 属性的值取决于这个属性先前被访问的次数：

```
class counter
{
    private int next;

    public int next {
        get { return next++; }
    }
}
```

因此，访问这个属性就产生一个可见的副效应，该属性应该作为一个方法而被执行。

get 访问函数的“无副效应”规则并不是说 get 访问函数应该总是只返回存储在域中的值。实际上，get 访问函数经常通过访问多个域或引用方法来计算一个属性的值。然而，一个正确设计的 get 访问函数不执行导致这个对象状态中的可见变化发生的操作。

属性可用于延迟一个援助的初始化过程直到这个援助第一次被引用。例如：

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;
    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(File.OpenStandardInput());
            }
            return reader;
        }
    }
    get {
        if (writer == null) {
            writer = new StreamWriter(File.OpenStandardOutput());
        }
        return writer;
    }
}

public static TextWriter Error {
    get {
        if (error == null) {
            error = new StreamWriter(File.OpenStandardError());
        }
        return error;
    }
}
```

```

    }
}
}

```

其中, console 类包括三个属性: in, out 和 error, 分别表示标准输入, 输出和错误装置。通过把这些成员作为属性来显示, 这个 console 类可延迟它们的初始化直到它们被实际应用。例如, 在对 out 属性的第一次引用中, 如: console, out, writeline (“hello, world”); 产生了这个输出方法的潜在的 textwriter。但是, 如果这个应用不引用 in 和 error 属性, 则不产生那些方法的对象。

10.6.3 虚拟、封装、重载和抽象访问函数 (Virtual, Sealed, Override And Abstract Accessors)

属性声明可能包括 static、virtual、override 和 abstract 修改函数的一个有效组合。含有 override 修改函数的属性也可以含有 sealed 修改函数 (参见 10.5.5 节)。

除一种情况外, 修改函数 static、virtual、override 和 abstract 互相排斥。修改函数 abstract 和 override 可一起使用, 其作用是使抽象属性可以重载虚拟属性。virtual 属性声明规定这个属性的访问函数是虚拟的。修改函数 virtual 适用于一个读写属性的两个访问函数——不可能只有一个读写属性的访问函数是虚拟的。

一个 abstract 属性声明规定这个属性的访问函数是虚拟的, 但并不提供该访问函数的一个实际执行。而非抽象的派生类需要通过重载该属性为这些访问函数提供它们自己的执行。因为抽象属性声明的访问函数不提供实际的执行, 其 accessor-body 只包括一个分号。

既包括 abstract 修改函数, 又包括 override 修改函数的属性声明规定那个属性是抽象的并重载一个基本属性。这样一个属性的访问函数也是抽象的。

抽象属性声明只允许出现在抽象类中。在派生类中, 继承虚拟属性的访问函数可通过含有指定一个 override 指令的属性声明而被重载。这称为 overriding properly declaration。重载属性声明不声明新属性。而只是特化一个现存虚拟属性的访问函数的执行。

重载属性声明必须指定与继承属性完全相同的可访问性修改函数、类型和名字。如果继承属性只有一个访问函数 (即如果这个继承属性是只读或只写的), 则重载属性可以且必须只包括那个访问函数。如果继承属性包括两个访问函数 (即如果这个继承属性是读写式的), 则重载属性既可以包括一个访问函数, 也可以包括两个访问函数。

重载属性声明可以含有 sealed 修改函数。这个修改函数的作用是阻止派生类进一步重载这个属性。封装属性的访问函数也是封装的。在重载属性声明中含有 new 修改函数是错误的。

除声明过程和引用语句不同之外, 虚拟、封装、重载及抽象访问函数与虚拟、封装、重载及抽象方法的作用是一致的。特别地, 在 10.5.3 节、10.5.4 节、10.5.5 节中所描述的规则的应用中, 这两个访问函数就相当于一个相应形式的方法:

- get 访问函数与一个无参数的方法是一致的, 这个方法具有属性类型的一个返回值, 且其修改函数与包含它的属性相同。
- set 访问函数与一个方法一致, 这个方法具有属性类型的一个值参数, 返回类型为 void, 且与包含它的属性具有相同的修改函数。

在下例中 x 是一个虚拟的只读属性, y 是一个虚拟的读写属性, z 是一个抽象的读写属性:

```
abstract class a
{
    int y;

    public virtual int x {
        get { return 0; }
    }

    public virtual int y {
        get { return y; }
        set { y = value; }
    }

    public abstract int z { get; set; }
}
```

因为 z 是抽象的, 所以包含它的类 a 也必须被声明为抽象的。

由 a 派生的一个类见下例:

```
class b: a
{
    int z;

    public override int x {
        get { return base.x + 1; }
    }

    public override int y {
        set { base.y = value < 0? 0: value; }
    }

    public override int z {
        get { return z; }
        set { z = value; }
    }
}
```

这里, x、y 和 z 的声明是重载属性声明。每一个属性声明都与相应的继承属性的可访问性修改函数、类型和名字完全匹配。x 的 get 访问函数和 y 的 set 访问函数用关键词 base 访问继承访问函数。z 的声明重载两个抽象的访问函数---因此, 在 b 中没有明显的抽象函数成员, 且 b 允许非抽象。

10.7 事件 (Events)

event 指的是能使对象或类提供通知的成员。通过提供 event handlers, 用户可以为事件联系可执行的代码。事件是用 event-declarations 声明的:

```

event-declaration:
    attributesopt event-modifiersopt event type
    variable-declarators
    attributesopt event-modifiersopt event type member-name
    { event-accessor-declarations }

event-modifiers:
    event-modifier
    event-modifiers event-modifier

event-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract

event-accessor-declarations:
    add-accessor-declaration remove-accessor-declaration
    remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:
    attributesopt add block

remove-accessor-declaration:
    attributesopt remove block

```

一个 `event-declaration` 可能包括一系列的 `attributes`、一个 `new` 修改函数、四个访问修改函数的有效组合及修改函数 `static virtual override` 和 `abstract` 的一个有效组合。另外，含有修改函数 `override` 的事件也可以含有修改函数 `sealed`。

除一种情况外，修改函数 `static virtual override` 和 `abstract` 互相排斥。修改函数 `abstract` 和 `override` 可一起使用，起作用的是使抽象事件可以重载虚拟事件。

一个事件声明可能含有 `event-accessor-declarations`，也可能由编译函数自动提供这样的访问函数。

省略 `event-accessor-declarations` 的事件声明定义一或多个事件——每一个都与一个 `variable-declarator` 相对应。这些属性和修改函数适用于所有由 `event-declaration` 声明的成员。

抽象事件的声明省略 `event-accessor-declarations` 的 `event-declaration`。`event-declaration` 不能既含有修改函数 `abstract`，又含有 `event-accessor-declarations`。

事件声明的类型必须是 `delegate-type`，且该 `delegate-type` 的可访问性必须至少与这个事件本身相同。

事件可被用作操作符的左操作数。这些操作符的作用是给一个事件连接事件处理函数，或把事件处理函数从这个事件中除去，这个事件的访问修改函数控制允许这个操作发生的上下文。

由于“+=”和“-=”是在声明这个事件的类型之外的事件中允许出现的唯一的操作，因此，外部代码可以增加或取消一个事件的处理函数，但不能以其它任何方式得到或修改事件处理函数的潜在列表。

在含有事件声明过程的类或结构的程序文内，某些事件可被当作域来使用。以这种方式被利用的事件不可能是抽象的，也不能明显地含有 event-accessor-declarations。这样的事件可用于允许域出现的任何上下文中。在下例中：

```
public delegate void eventhandler(object sender, eventargs e);

public class button: control
{
    public event eventhandler click;

    protected void onclick(eventargs e) {
        if (click != null) click(this, e);
    }

    public void reset() {
        click = null;
    }
}
```

在 button 类中，click 被用作一个域。如此例所示，这个域可被检查、修改以及用在委托引用表达式中。此 button 类中的方法 onclick “提出”这个 click 事件。提出一个事件的意思就是引用由这个事件所表示的委托——因此，提出事件没有专门的语言结构。注意，委托引用之前有一个检查以确保这个委托是非 null 的。

在 button 类声明过程之外，成员 click 只能被用在操作符左边，如下例中：

```
b.click += new eventhandler(...);
```

它把一个委托附加到这个 click 事件的引用列表中，而下例则把一个委托从这个 click 事件的引用列表中除去。

在式子 x+=y 或 x-=y 的操作中，当 x 是一个事件，且引用发生在含有 x 声明的类型之外时，这个操作的结果就是 void（与赋值后 x 的值相反）。这条规则禁止外部代码间接检测一个事件的潜在委托。

下例表明事件处理函数怎样被连接到上述 button 类的实例中：

```
public class logindialog: form
{
    button okbutton;
    button cancelbutton;

    public logindialog() {
        okbutton = new button(...);
        okbutton.click += new eventhandler(okbuttonclick);
    }
}
```

```

cancelbutton = new button(...);
cancelbutton.click += new eventhandler(cancelbuttonclick);
}

void okbuttonclick(object sender, eventargs e) {
    // handle okbutton.click event
}

void cancelbuttonclick(object sender, eventargs e) {
    // handle cancelbutton.click event
}

```

这里构造函数 login dialog 产生两个 button 实例, 并把事件处理函数连接到 click 事件中。

10.7.1 事件访问函数 (Event Accessors)

如在上面 button 例中所示, 典型的事件声明省略 event-accessor-declarations。当事件域的存储费用不可接受时, 类可含有 event-accessor-declarations, 并用一秘密方法存储事件处理函数的列表。

一个事件的 event-accessor-declarations 规定与添加或取消事件处理函数有关的可执行语句。

访问函数声明包括一个 add-accessor-declaration 和 remove-accessor-declaration。每一个访问函数声明都是由符号 add 或 remove 及一个 block 组成的。与 add-accessor-declaration 有关的 block 规定, 添加一个事件处理函数要执行的语句, 而与 remove-accessor-declaration 有关的 block 则规定取消一个事件处理函数要执行的语句。

事件访问函数, 不管是 add-accessor-declaration 还是 remove-accessor-declaration, 都与一个方法一致, 这个方法具有这个事件类型的一个单值参数且返回类型为 void。一个事件访问函数的默认参数总被命名为 value, 当一个事件被用在事件赋值过程中时, 所用的事件访问函数一定要合适。如果这个赋值操作的操作符是 +=, 那么就用添加访问函数, 而如果这个赋值操作符是 -=, 那么就用取消访问函数。在其它情况下, 赋值操作符的右边作为事件访问函数的自变量。一个 add-accessor-declaration 或 remove-accessor-declaration 的块必须遵守 10.5.8 节中所述的方法 void 的规则。尤其是不允许这样一个块中的 return 语句指定一个表达式。

由于事件访问函数固定具有名字为 value 的一个参数, 因此, 事件访问函数中的局部变量声明不能使用该名字, 否则就是错误的。

在下例中类 control 执行事件内部存储的一个方法:

```

class control: component
{
    // unique keys for events
    static readonly object mousedowneventkey = new object();
    static readonly object mouseupeventkey = new object();
    // return event handler associated with key
    protected delegate geteventhandler(object key) {...}
}

```

```

// add event handler associated with key
protected void addeventhandler(object key, delegate handler) {...}
// remove event handler associated with key
protected void removeeventhandler(object key, delegate handler) {...}
// mousedown event
public event mouseeventhandler mousedown {
    add { addeventhandler(mousedowneventkey, value); }
    remove { addeventhandler(mousedowneventkey, value); }
}
// mouseup event
public event mouseeventhandler mouseup {
    add { addeventhandler(mouseupeventkey, value); }
    remove { addeventhandler(mouseupeventkey, value); }
}
}

```

方法 `addeventhandler` 把一个委托值与一个键联系起来，方法 `get event handler` 返回当前与一个键有关的委托，而方法 `remove event handler` 则作为特定事件的事件处理函数取消一个委托。如果设计一个潜在的存储方法，使得把一个委托值 `null` 与一个键联系起来时不需要任何代价，则未处理事件也就没有存储函数。

注意：在 Microsoft .net 运行期间，当一个类声明类型 `t` 的一个属性 `x` 时，它不能再声明具有下列签名的方法，否则就是错误的：

```

void add_x(t handler);
void remove_x(t handler);

```

Microsoft .net 运行期为相应的程序语言保留这些签名，这些程序语言在连接或取消事件处理函数时不提供操作符或其它语言结构。注意，这并不意味着在 `c#` 程序中可以用方法语句连接或取消事件处理函数。它的意思只是，具有这种模式的属性和方法不能出现在同一各类中。

当类声明一个事件时，`c#` 编译函数就自动产生上面所述的方法 `add_x` 和 `remove_x`。例如，下面的声明：

```

class button
{
    public event eventhandler click;
}

```

可看作：

```

class button
{
    private eventhandler click;
    public void add_click(eventhandler handler) {

```



```

        click += handler;
    }

    public void remove_click(eventhandler handler) {
        click -= handler;
    }
}

```

编译函数进一步产生一个引用方法 `add_x` 和 `remove_x` 的事件。从 `c#` 程序方面来看，这些操作都是纯粹的执行过程的细节，除符号 `add_x` 和 `remove_x` 被保留外，没有其它可见的效果。

10.7.2 静态事件 (Static Events)

当一个事件声明含有修改函数 `static` 时，这个事件就是一个 **static event**。当没有 `static` 修改函数时，这个事件就是一个 **instance event**。

静态事件与具体的实例无关，且在静态事件的访问函数中不能引用 `this`，否则，就是错误的。而且，在静态事件中含有修改函数 `virtual abstract` 或 `override` 也是错误的。

实例事件与一个类的给定实例有关，且在这个事件的访问函数中，这个实例可作为 `this` 被访问。

当一个事件在式子 `e.m` 的 **member-access** 中被引用时，如果 `m` 是一个静态事件，则 `e` 必须表示一个类型；如果 `m` 是一个实例事件，则 `e` 必须表示一个实例。

静态和实例成员的不同点在 10.2.5 节中有进一步探讨。

10.8 索引 (Indexers)

一个 **indexers** 就是一个成员，它使得一个对象以与数组相同的方式被索引。索引用 **indexer-declarations** 来声明：

```

indexer-declaration:
    attributesopt  indexer-modifiersopt  indexer-declarator
    {  accessor-declarations  }

indexer-modifiers:
    indexer-modifier
    indexer-modifiers  indexer-modifier

indexer-modifier:
    new
    public
    protected
    internal
    private
    virtual
    sealed
    override
    abstract

```

```
indexer-declarator:  
type this [ formal-parameter-list ]  
type interface-type this [ formal-parameter-list ]
```

一个 `indexer-declaration` 可能包括一系列 `attributes`、一个 `new` 修改函数、四个访问修改函数的有效组合及修改函数 `virtual`、`override` 和 `abstract` 的一个有效组合。另外，含有 `override` 修改函数的一个索引也可以含有 `sealed` 修改函数。

除一种情况外，修改函数 `static`、`virtual`、`override` 和 `abstract` 互相排斥。`abstract` 和 `override` 可一起使用，其结果是使抽象索引可以重载虚拟索引。

索引声明的 `type` 规定由这个声明所引入的索引的元素类型。除非这个索引是一个显式接口成员执行。否则，这个 `type` 后就跟关键词 `this`。对于一个显式接口成员执行来说，`type` 后是一个 `interface-type`、“.”和关键词 `this`。与其它成员不同，索引没有用户自定义名。

`formal-parameter-list` 规定索引的参数。除非必须指定至少一个参数或 `ref` 和 `out` 参数修改函数不被允许，否则，索引的形参列表与方法形参列表是一致的。

一个索引的 `type` 及在 `formal-parameter list` 中引用的每一个类型都必须至少具有与这个索引本身相同的可访问性。

必须包含在符号“{”和“}”内的 `accessor-declarations` 声明这个索引的访问函数。这些访问函数规定与读和写索引元素有关的可执行语句。

即使访问一个索引元素的句法与访问一个数组元素的句法相同，索引元素也不是变量。因此，把一个索引元素当作一个 `ref` 或 `out` 参数来传递是错误的。

一个索引的形参列表定义这个索引的签名。索引的签名包括其形参的数量及类型。元素类型不是索引签名的一部分，形参名也不是。

一个索引的签名必须与在同一类中声明的所有其它索引的签名不同。

索引与属性从概念上非常相似，但在以下几方面不同：

- 属性由其名来标识，而索引则用签名来标识。
- 属性通过 `simple-name` 或 `member-access` 来访问，而索引成员则通过 `element-access` 来访问。
- 属性可以是一个 `static` 成员，而索引则总是一个实例成员。
- 属性的 `get` 访问函数与一个无参数的方法一致，而索引的 `get` 访问函数则与一个同这个索引具有相同形参列表的方法一致。
- 属性的 `set` 访问函数与具有一个命名为 `value` 的参数的方法一致；而索引的 `set` 访问函数则与下面的方法一致。这些方法与该索引具有相同的形参列表，另外还有一个名为 `value` 的额外参数。
- 索引访问函数声明一个同某一索引参数具有相同名字的局部变量是错误的。
- 在重载属性声明中，继承属性用句法 `base.p` 来访问，这里，`p` 是该属性名。在重载索引声明过程中，继承的索引引用句法 `base[e]` 来访问，这里，`e` 是一个用逗号隔开的表达式列表。

由于这些不同点，在 10.6.2 节和 10.6.3 节中定义的所有规则既适用于属性访问函数，也适用于索引访问函数。

注意: 在 Microsoft .net 运行期, 当一个类声明一个带有形参列表 p 的类型 t 的索引函数时, 它不能在声明一个具有下列签名的方法:

```
t get_item(p);
void set_item(p, t value);
```

Microsoft .net 运行期不支持索引函数的程序语言保留这些签名。注意, 这并不意味着 C# 程序可以用方法语句访问索引函数或用索引语句访问方法。它的意思只是, 具有这种模式的索引函数和方法不能出现在同一各类中。

下在的例子声明一个 `bitarray` 类, 这个类执行访问位数组中单个位的一个索引:

```
class bitarray
{
    int[] bits;
    int length;
    public bitarray(int length) {
        if (length < 0) throw new argumentexception();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }
    public int length {
        get { return length; }
    }
    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new indexoutofrangeexception();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new indexoutofrangeexception();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```

```

}
}

```

类 `bitarray` 的一个实例所用的存储空间比相应的 `bool[]` 要少得多 (每一个值只占一个位而不是一个字节的位置), 但它允许与 `bool[]` 相同的操作。

下面的 `countprimes` 类用一个 `bitarray` 及传统的“筛分”规则计算中心和给定的最大值之间的素数数量。

```

class countprimes
{
    static int count(int max) {
        bitarray flags = new bitarray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void main(string[] args) {
        int max = int.parse(args[0]);
        int count = count(max);
        console.writeline("found {0} primes between 1 and {1}", count, max);
    }
}

```

注意: 访问 `bitarray` 的元素的句法与访问 `bool[]` 的句法完全相同。

索引重载分解规则如 7.4.2 节中所述。

10.9 操作符 (Operators)

一个 `operator` 就是一个成员。这个成员定义表达式操作符的含义, 这样的操作符适用于类的实例。操作符用 `operator-declarations` 来声明:

```

operator-declaration:
    attributesopt operator-modifiers operator-declarator block
operator-modifiers:
    public static
    static public
operator-declarator:
    unary-operator-declarator

```

```

binary-operator-declarator
conversion-operator-declarator
unary-operator-declarator:
type operator overloadable-unary-operator ( type identifier )
overloadable-unary-operator: one of
+ - ! ~ ++ -- true false
binary-operator-declarator:
type operator overloadable-binary-operator ( type identifier
type identifier )
overloadable-binary-operator: one of
+ - * / % & ; ^ << >> == != > < >= <=
conversion-operator-declarator:
implicit operator type ( type identifier )
explicit operator type ( type identifier )

```

有三种操作符：一元操作符，二元操作符和转换操作符。

下面的规则适用于所有的操作符声明：

- 操作符声明必须既包括 **public** 修改函数，又包括 **static** 修改函数，且不允许含有任何其它修改函数。
- 一个操作符的参数必须是值参数。操作符声明不能指定 **ref** 或 **out** 参数，否则，就是错误的。
- 操作符的签名必须与在同一类中声明的所有其它操作符的签名不同。
- 操作符声明中所引用的所有类型都必须与这个操作符本身具有相同的可访问性。

每种操作符也都有其附加的规则，如以下几部分所述。与其它成员相似，在基类中声明的操作符被派生类继承。由于操作符声明总是要求声明它的类或结构参与其签名，因此，在派生类中声明的操作符不可能隐藏在基类中声明的操作符。所以，在操作符声明中，既不需要，也不允许有 **new** 修改函数。对于所有操作符来说，操作符声明都包括一个 **block**，这个 **block** 指明引用该操作符时要执行的语句。操作符的块必须遵守 10.5.8 节中所述的关于值返回方法的规则。

关于一元和二元操作符的其它信息见 7.2 节。关于转换操作符的其它信息见 6.4 节。

10.9.1 一元操作符 (Unary Operators)

下面的规则适用于一元操作符声明，这里，**t** 表示含有操作符声明的类或结构类型：

- 一元操作符 **+**，**-**，**!** 或 **~** 必须使用类型 **t** 的一个参数，且能返回任何类型。
- 一元操作符 **++** 或 **--** 必须使用类型 **t** 的一个参数，且必须返回类型 **t**。
- 一元操作符 **true** 或 **false** 必须使用类型 **t** 的一个参数，且必须返回类型 **bool**。

一元操作符的签名包括操作符符号及单个形参的类型。返回类型不是一元操作符签名的部分，形参的名字也不是。

一元操作符 **true** 和 **false** 必须成对声明。如果一个类只声明其中的一个而不声明另一个，则将出现错误。操作符 **true** 和 **false** 在 7.16 节中有进一步描述。

10.9.2 二元操作符 (Binary Operators)

一个二元操作必须使用两个参数，其中至少有一个来自于该操作符的类或结构中的。二元操作符可返回任何类型。

二元操作符的签名包括操作符符号及两个形参的类型。返回类型不是二元操作符签名的一部分，形参名也不是。

某些二元操作符需要成对声明。对于两个操作符声明来说，必须有其中的另一个操作符的匹配声明。当两个操作符声明具有相同的返回类型且每一个参数的类型也相同时，这两个操作符声明就匹配。下列操作符需要成对声明：

- 操作符 `=` 和 `!=`
- 操作符 `>` 和 `<`
- 操作符 `>=` 和 `<=`

10.9.3 转换操作符 (Conversion Operators)

转换操作符声明引入一个 user-defined conversion，这个 user-defined conversion 扩增预先定义的隐式和显式转换。

含有关键词 `implicit` 的转换操作符声明引入一个用户自定义隐式转换。隐式转换可在多种情况下发生，包括函数成员引用、`cast` 表达式和赋值。这在 6.1 节中有进一步描述。

含有关键词 `explicit` 的转换操作符声明引入一个用户自定义显式转换。显式转换可在 `cast` 表达式中发生，在 6.2 节中有进一步描述。

一个转换操作符把原类型转化为目标类型，原类型即这个转换操作符的参数类型，目标类型即这个转换操作符的返回类型。如果下列条件都符合，则类或结构就允许声明从原类型 `s` 转换到目标类型 `t`：

- `s` 和 `t` 的类型不同
- `s` 和 `t` 有一个是声明操作符的类或结构类型。
- `s` 和 `t` 都不是 `object` 或 `interface-type`。
- `t` 不是 `s` 的基类，`s` 也不是 `t` 的基类。

从第一条规则可知，转换操作符必须转化或者声明这个操作符的类或结构类型或者从这个类或结构类型转化为其它类型。例如，类或结构类型 `c` 可以定义一个从 `c` 到 `int` 的转换，也可以定义一个从 `int` 到 `c` 的转换，但不能定义一个从 `int` 到 `bool` 的转换。

重新定义一个预先定义的转换是不可能的。因此，转换操作符不允许从 `object` 转化为 `object`，因为在 `object` 和所有其它类型之间已经存在隐式和显式转换。同样，一个转换的原类型和目标类型都不能是彼此的基类，因为在它们之间已经存在一个转换。

用户自定义转换不允许转化为 `interface-type` 或由它转化为其它类型。这条规则确保转化为 `interface-type` 时不会发生用户自定义转换，并确保只有当正在被转化的对象真的执行指定的 `interface-type` 时，到 `interface-type` 的转换才能成功。

转换操作符的签名包括原类型和目标类型。（注意，这是返回类型参加签名的成员的唯一形式）。转换操作符的 `implicit` 或 `explicit` 分类不是操作符签名的一部分。因此，类或结构

不能同时声明具有相同原类型和目标类型的一个 `implicit` 和 `explicit` 转换操作符:

一般而言, 所设计的用户自定义隐式转换应该既不抛出异常, 也不丢失信息。如果一个用户自定义转换可能产生异常 (例如, 由于原自变量超出了取值范围) 或丢失信息 (如丢失高序位), 则那个转换就应被定义为一个显式转换。

在下例中:

```
public struct digit
{
    byte value;
    public digit(byte value) {
        if (value < 0 || value > 9) throw new argumentexception();
        this.value = value;
    }
    public static implicit operator byte(digit d) {
        return d.value;
    }
    public static explicit operator digit(byte b) {
        return new digit(b);
    }
}
```

从 `digit` 到 `byte` 的转换是隐式的, 因为它从不抛出异常或丢失信息, 但从 `byte` 到 `digit` 的转换是显式的, 因为 `digit` 只能表示 `byte` 的可能值的子集。

10.10 实例构造函数 (Instance Constructors)

一个 `instance constructor` 就是一个成员, 这个成员执行初始化一个类的实例时所需要的操作。构造函数用 `constructor-declarations` 来声明。

```
constructor-declaration:
    attributesopt constructor-modifiersopt constructor-declarator block
constructor-modifiers:
    constructor-modifier
constructor-modifiers constructor-modifier
constructor-modifier:
    public
    protected
    internal
    private
constructor-declarator:
    identifier ( formal-parameter-listopt ) constructor-initializeropt
constructor-initializer:
    : base ( argument-listopt )
```

```
: this ( argument-listopt )
```

一个 **constructor-declaration** 可能包括一系列 **attributes** 及四个访问修改函数的一个有效组合。

一个 **constructor-declaration** 的 **identifier** 必须是声明这个构造函数的名字。如果指定任何其它名字, 都会发生错误。

一个构造函数的可选的 **formal-parameter-list** 同方法的 **formal-parameter-list** 遵守相同的规则。形参列表定义构造函数的签名并控制一个过程, 通过这个过程, 重载分解选择一个引用中特定的构造函数。

在一个构造函数的 **formal-parameter-list** 中引用的每一个类型都必须至少具有与这个构造函数本身相同的可访问性。

可选的 **constructor-initializes** 指定要引用的构造函数, 这个引用发生在执行该构造函数的 **block** 中所给定的语句之前。这在 10.10.1 中有进一步描述。

一个构造函数声明的 **block** 规定初始化类的一个新实例要执行的语句。这与返回值为 **void** 的实例方法的 **block** 完全一致。

构造函数不是继承的。因此, 一个类除那些在其中实际声明的构造函数外, 没有其它构造函数。如果一个类没有构造函数声明, 则系统自动提供一个默认的构造函数。

构造函数通过 **constructor-initializers** 被 **object-creation expression** 引用。

10.10.1 构造函数初始化 (Constructor Initializers)

除类 **object** 的构造函数外, 所有的构造函数都固定含有另一个构造函数的一个引用, 后者直接在这个构造函数的 **block** 中的第一条语句之前。被隐式引用的构造函数取决于这个 **construct-initializer**:

- 式子 **base()** 的构造函数初始化函数产生一个来自于要引用的直属基类的构造函数。构造函数用 7.4.2 节中的重载分解规则选择。候选构造函数的集合包括在这个直属基类中声明的所有可访问的构造函数。如果候选构造函数的集合是空的, 或者不能识别一个单个最佳构造函数, 则错误出现。
- 式子 **this()** 的构造函数初始化函数产生一个来自于要引用的类本身的构造函数。这个构造函数用 7.4.2 节中的重载分解规则选择。候选构造函数的集合包括所有在该类中声明的可访问的构造函数。如果候选构造函数的集合是空的, 或者不能识别一个单个最佳构造函数, 则错误发生。如果一个构造函数声明包括引用这个构造函数本身的构造函数初始化, 则错误出现。

如果一个构造函数没有构造函数初始化函数, 则式子 **base()** 默认提供一个构造函数的初始化函数。因此, 式子 **c (...) {...}** 的构造函数声明就相当于 **c (...): base () {...}**。

由构造函数声明的 **formal-parameter-list** 给定的参数范围包括该声明的构造函数初始化函数。因此, 一个构造函数初始化函数允许访问这个构造函数的参数。例如:

```
class a
{
    public a(int x, int y) {}
```



```

}
class b: a
{
    public b(int x, int y): base(x + y, x - y) {}
}

```

构造函数初始化函数不能访问正产生的实例。因此，在构造函数初始化函数的自变量表达式中引用 `this` 是错误的，因为自变量表达式通过 `simple-name` 引用任何实例成员都是错误的。

10.10.2 实例变量初始化函数 (Instance Variable Initializers)

当一个构造函数没有构造函数初始化函数或式子 `base(...)` 的构造函数初始化函数时，这个构造函数就默认执行由在这个类中声明的实例域的 `variable-initializers` 指定的初始化。这与一系列赋值一致，这些赋值在该构造函数的入口即被执行，且在其直属基类构造函数隐式引用之前进行。变量初始化函数以它们在类声明中出现的原文顺序被执行。

10.10.3 构造函数执行 (Constructor Execution)

把实例变量初始化函数及构造函数初始化函数看作这样的语句是有意义的，这些语句在构造函数的 `block` 中的第一条语句之前就自动被插入。下面的例子：

```

using system.collections;

class a
{
    int x = 1, y = -1, count;

    public a() {
        count = 0;
    }

    public a(int n) {
        count = n;
    }
};

class b: a
{
    double sqrt2 = math.sqrt(2.0);
    arraylist items = new arraylist(100);
    int max;

    public b(): this(100) {
        items.add("default");
    }
}

```

```
public b(int n): base(n - 1) {  
    max = n;  
}  
}
```

包括几个变量初始化函数,也包括两种式子(base 和 this)的构造函数初始化函数。这个例子与下面所示的代码一致,这里,每一个注释都表示一个自动插入语句用于自动插入的构造函数引用的句法是无效的,只是为了说明这个方法。

```
using system.collections;  
  
class a  
{  
    int x, y, count;  
  
    public a() {  
        x = 1; // variable initializer  
        y = -1; // variable initializer  
        object(); // invoke object() constructor  
        count = 0;  
    }  
  
    public a(int n) {  
        x = 1; // variable initializer  
        y = -1; // variable initializer  
        object(); // invoke object() constructor  
        count = n;  
    }  
}  
  
class b: a  
{  
    double sqrt2;  
    arraylist items;  
    int max;  
  
    public b(): this(100) {  
        b(100); // invoke b(int) constructor  
        items.add("default");  
    }  
  
    public b(int n): base(n - 1) {  
        sqrt2 = math.sqrt(2.0); // variable initializer
```

```

        items = new arraylist(100);           // variable initializer
        a(n - 1);                             // invoke a(int) constructor
        max = n;
    }
}

```

注意：变量初始化函数被转化为赋值语句，这些赋值语句在基类构造函数引用之前被执行。这个命令确保所有的实例域被它们的变量初始化函数初始化，时间是在任何可访问该实例的语句被执行之前。例如：

```

class a
{
    public a() {
        printfields();
    }

    public virtual void printfields() {}
}

class b: a
{
    int x = 1;
    int y;

    public b() {
        y = -1;
    }

    public override void printfields() {
        console.writeline("x = {0}, y = {1}", x, y);
    }
}

```

当用 `new b()` 产生 `b` 的一个实例时，输出：`x=1, y=0`

`x` 的值是 1，因为这个变量初始化函数在基类构造函数被引用之前被执行。然而，`y` 的值是 0，因为对 `y` 的赋值直到这个基类构造函数返回之后才被执行。

10.10.4 默认构造函数 (Default Constructors)

如果一个类不含有构造函数声明，则系统就自动提供一个默认的构造函数。默认的构造函数只引用其直属基类的无参数构造函数。如果其直属基类没有可访问的构造函数，则错误发生。如果这个类是抽象的，则默认构造函数所声明的可访问性就被保护。否则，默认构造函数声明的可访问性就是公开的。因此，默认构造函数的形式总是，`protected c()`。

这里，`c` 就是这个类的名字。

下例中提供了一个默认构造函数:

```
class message
{
    object sender;
    string text;
};
```

因为这个类不含有构造函数声明。因此, 这个例子就相当于:

```
class message
{
    object sender;
    string text;

    public message(): base() {}
}
```

10.10.5 局部构造函数 (Private Constructors)

当一个类只声明局部构造函数时, 其它的类就不可能由这个类派生而来, 也不可能产生这个类的实例 (嵌套在这个类内的一个异常)。局部构造函数通常只用在只含有静态成员的类中。例如:

```
public class trig
{
    private trig() {}          // prevent instantiation

    public const double pi = 3.14159265358979323846;

    public static double sin(double x) {...}
    public static double cos(double x) {...}
    public static double tan(double x) {...}
}
```

类 `trig` 提供一组相关的方法和常量, 但不能被例示。因此, 它只声明一单个局部构造函数。为了阻止默认构造函数的自动产生, 必须声明至少一个局部构造函数。

10.10.6 可选的构造函数参数 (Optional Constructor Parameters)

构造函数的初始化函数的式子 `this()` 通常与重载一块儿使用以执行可选的构造函数参数。下例中最初的两个构造函数只为忽略的变量提供默认值:

```
class text
{
    public text(): this(0, 0, null) {}
}
```

```

public text(int x, int y): text(x, y, null) {}
public text(int x, int y, string s) {
    // actual constructor implementation
}
}

```

它们都用一个 `this()` 构造函数的初始化函数引用第三个构造函数，其作用实际上是初始化这个新的实例。其效果就是可选择的构造函数参数的结果：

```

text t1 = new text();           // same as text(0, 0, null)
text t2 = new text(5, 10);      // same as text(5, 10, null)
text t3 = new text(5, 20, "hello");

```

10.11 静态构造函数 (Static Constructors)

一个 **static constructor** 就是一个成员，这个成员执行初始化一个类时所需的操作。静态构造函数用 `static-constructor-declarations` 来声明：

```

static-constructor-declaration:
    attributesopt static identifier ( ) block

```

一个 `static-constructor-declaration` 可能含有一系列 `attributes`。

一个 `static-constructor-declaration` 的 `identifier` 必须是声明这个静态构造函数的类的名字。作为任何其它名字都是错误的。

一个静态构造函数声明的 `block` 规定初始化这个类时要执行的语句。这与返回类型为 `void` 的静态方法的 `block` 一致。

静态构造函数是不继承的。

- **class loading** 就是一个过程，它为运行期环境提供一个类。尽管有一些保证，但登录过程绝大多数取决于执行过程。
- 一个类在其任何实例产生之前登录。
- 一个类在其静态成员被引用之前登录。
- 一个类在由它派生的任何类型登录之前登录。
- 在一个程序的一个执行过程中，一个类的登录不能超过一次。
- 如果一个类有一个静态构造函数，那么，当这个类登录时，它就自动被调用。静态构造函数不能被显式引用。下列：

```

class test
{
    static void main() {
        a.f();
        b.f();
    }
}

```

```
class a
{
    static a() {
        console.WriteLine("init a");
    }
    public static void f() {
        console.WriteLine("a.f");
    }
}
class b
{
    static b() {
        console.WriteLine("init b");
    }
    public static void f() {
        console.WriteLine("b.f");
    }
}
```

或输出

```
init a
a.f
init b
b.f
```

或输出

```
init b
init a
a.f
b.f
```

因为登录也即静态构造函数执行的确切顺序还未确定。

下例:

```
class test
{
    static void main() {
        console.WriteLine("1");
        b.g();
        console.WriteLine("2");
    }
}
```

```

    }
    class a
    {
    static a() {
        console.WriteLine("init a");
    }
    }
    class b: a
    {
    static b() {
        console.WriteLine("init b");
    }
    public static void g() {
        console.WriteLine("b.g");
    }
    }
}

```

保证输出:

```

init a
init b
b.g

```

因为类 a 的静态构造函数必须在类 b 的静态构造函数执行之前进行, 这里, b 是由 it 派生而来的。

这样, 就可能形成循环依赖关系, 它允许具有变量初始化函数的静态域以它们的默认值状态出现。

下面的例子:

```

class a
{
    public static int x = b.y + 1;
}
class b
{
    public static int y = a.x + 1;
    static void main() {
        console.WriteLine("x = {0}, y = {1}", a.x, b.y);
    }
}

```

输出:

```
x=1, y=2
```

为执行方法 `main`，系统首先要登录类 `b`。`b` 的静态构造函数计算 `y` 的初始值，这个过程同时也登录了 `a`，因为其中引用了 `ax` 的值。`a` 的静态构造函数反过来又计算 `x` 的初始值，且同时又推导出 `y` 的 default 值 0。这样，`ax` 就被初始化为 1。登录 `a` 的过程也就结束了，同时又把 `y` 的值返回到计算过程中，其结果为 `z`。

如果 `main` 方法在类 `a` 中登录，则此例将输出：

```
x=2, y=1
```

应避免静态域初始化函数的循环引用，因为它不能决定含有类引用的类的登录顺序。

10.12 析构函数 (Destructors)

一个 destructor 就是一个成员，这个成员执行析构某类的实例时所需要的操作。析构函数用 `destructor-declarations` 声明：

```
destructor-declaration:
    attributesopt ~ identifier ( ) block
```

一个 `destructor-declaration` 可能含有一系列 `attributes`。

一个 `destructor-declaration` 的 `identifier` 必须是声明析构函数的类的名字。如果指定任何其它名字，则错误发生。

一个析构函数声明的 `block` 规定析构这个类的一个实例时要执行的语句。这与返回值为 `void` 的实例方法的 `block` 完全一致（参见 10.5.8 节）。

析构函数是不继承的。因此，除在其中实际声明的外，一个类没有其它析构函数。

析构函数自动被引用，且不能被显式引用。合格的实例析构指的是经析构的实例再也不能被任何代码所用。实例析构函数的执行发生在这个实例适合析构后的任何时间。当一个实例被析构时，在继承链中的析构函数按顺序被调用，从最原始的到最外层的。

第 11 章 结 构

结构和类相似。它们都是数据结构，包含数据成员和函数成员。与类不同的是，结构是值类型且不需要堆分配。一个结构型的变量直接包含该结构的数据，而类类型的变量只是该数据的一个引用，后者即对象。

结构对于那些含有值语义的小型数据结构尤其有用。复数、坐标系中的点以及字典中关键值对等等都是结构的极好例子。数据结构的特点就在于，它的数据成员很少，不必使用继承或完全等同引用；它可以方便的应用值语义，赋值时直接复制值而非引用。

如 4.1.3 节中所述，C#所提供的简单类型，如 `int`、`double`、`bool` 等，实际上都是结构类型。由于这些预处理类型是结构，因此，C#语言中可用结构和操作符重载执行新的“简单”类型。该类型的两个例子将在本章节尾 11.4 节中给出。

11.1 结构声明 (Struct Declarations)

一个 `struct-declaration` 是声明一个新结构的 `type-declaration`(参见 9.5 节)：

```
struct-declaration:  
    attributesopt struct-modifiersopt struct identifier  
    struct-interfacesopt struct-body ;opt
```

一个 `struct-declaration` 包括可选择的一系列 `attributes` (参见 17 节), 然后是可选择的一系列 `struct-modifiers` (参见 11.1.1 节), 随后是关键字 `struct` 和一个命名结构的 `identifier`, 接着是可选的 `struct-interfaces` 说明 (参见 11.1.2 节), 然后一个 `struct-body` (参见 11.1.3 节), 最后是一个可选的分号。

11.1.1 结构修改函数 (Struct Modifiers)

一个 `struct-declaration` 包括可选择性的一系列结构修改函数：

```
struct-modifiers:  
struct-modifier  
struct-modifiers struct-modifier  
  
struct-modifier:  
new  
public  
protected  
internal  
private
```

11.1.2 结构接口 (Struct Interfaces)

一个结构声明可包括一个 `struct-interfaces` 说明, 用以指令结构执行给定的接口类型。

```
struct-interfaces:\  
    : interface-type-list
```

接口的应用将在 13.4 节作进一步讨论。

11.1.3 结构主体 (Struct Body)

结构的 `struct-body` 定义该结构的成员。

```
struct-body:  
{ struct-member-declarationsopt }
```

11.2 结构成员 (Struct Members)

一个结构的成员包括由它的 `struct-member-declaration` 引入的成员和由 `object` 类型继承的成员。

```
struct-member-declarations:  
struct-member-declaration  
struct-member-declarations struct-member-declaration  
struct-member-declaration:  
constant-declaration  
field-declaration  
method-declaration  
property-declaration  
event-declaration  
indexer-declaration  
operator-declaration  
constructor-declaration  
static-constructor-declaration  
type-declaration
```

除了 11.3 节中所述的差异, 10.2~10.11 节中对成员的描述同样适用于结构成员。

11.3 类和结构差异 (Class And Struct Differences)

11.3.1 值语义 (Value Semantics)

结构是值类型 (参见 4.1 节) 且有值语义。而类是引用类型, 具有引用语义。

结构类型的变量直接包括该结构的数据，而类类型的变量只是数据的引用；后者即对象。

在类中，两个变量可引用同一个对象；由此，对于一个变量的操作也可以影响到被另一变量所引用的对象。在结构中，每个变量都有其数据拷贝，对于一个变量的操作不会影响到另一个。另外，因为结构不是引用类型，结构类型的值不可能是 `null`。

给定下列声明：

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

其中的代码片段输出值 10。

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
Console.WriteLine(b.x);
```

a 到 b 的赋值产生了该值的一个拷贝，因此 a.x 的赋值对 b 没有影响。若 point 被声明为类，a 和 b 将引用同一个对象，因而输出将会是 100。

11.3.2 继承 (Inheritance)

所有结构类型都固定从类 `object` 中继承而来。一个结构声明可以指定执行的接口名单，但它不能指定基本类。

结构类型永远不会是抽象的，且总是隐式封箱。因而结构声明中不允许出现 `abstract` 和 `sealed` 修改函数。

由于继承不支持结构，所以结构成员声明的可访问性不能是 `protected` 或 `protected internal`。

结构函数成员不能是 `abstract` 或 `virtual`，且 `override` 修改函数 只能用来替换从 `object` 类型继承的方法。

11.3.3 赋值 (Assignment)

赋值结构类型的变量将产生所赋值的一个 `copy`。这不同于类类型变量的赋值。类类型变量的赋值是拷贝引用而不是由引用所标识的对象。

与赋值类似，当结构以值参数传递或以一个函数成员的结果返回时，该结构将建立一个拷贝。使用 `ref` 或 `out` 参数，结构可以在函数成员之间传递。

当结构的属性或索引作为赋值目标时，与其有关的实例表达式必须是变量。若将其归为一个值，编译时将会出现错误。这在 7.13.1 节中有进一步详述。

11.3.4 默认值 (Default Values)

如 5.2 节所述，有几种变量一经生成即被赋以默认值。对于类类型和其它引用类型的变量来说，其默认值为 `null`。然而，由于结构是值类型不能是 `null`，其默认值代以“清零”该结构的域时所产生的值。

参照上述 `point` 结构，下例数组中的每一个 `point` 被初始化为一个值，此值由清零 `x` 和 `y` 域产生：

```
Point[] a = new Point[100];
```

结构的默认值即相当于该结构默认构造函数（参见 4.1.1 节）时的返回值。与类不同，结构不能声明无参构造函数。但每一个结构固定拥有一个无参数的构造函数，且该构造函数返回值总是由清零结构的域所产生。

设计结构时须认为缺省初始化状态为有效状态。下例用户自定义构造函数只有在空值被显式调用时才受到保护：

```
struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

当 `KeyValuePair` 以默认值初始化为条件时，`Key` 及 `Value` 域将为零，这时，需准备结构处理这种情况。

11.3.5 封箱和非封箱 (Boxing And Unboxing)

编译时将引用视为另一类型的类即可将 `class` 类型的值转化为 `object` 类型或接口类型。同样，`object` 类型和接口类型的值也可以转回到 `class` 类型而不必改变引用（当然，该例中需运行时检查）。

由于结构非引用类型，结构类型操作的执行有所不同。在结构执行将结构类型的值转化为 `object` 类型或接口类型时，产生封箱操作。同样，在 `object` 类型和接口类型转化回结构类型时，产生非封箱操作。与 `class` 类型操作的主要差异在于封箱和非封箱 `copies` 出来或进入封箱实例的结构值。这样，封箱和非封箱操作之后，非封箱结构的改变不会反映到封箱结构。

更多描述请参阅参见 4.3 节。

11.3.6 this 的意思 (Meaning Of This)

在类的构造函数和实例函数中, this 表示一个值。这样, 当 this 表示引用函数成员的实例时, 在类函数成员中不可能对 this 赋值。

在结构构造函数中, this 相当于结构类型的一个 out 参数; 在结构的实例函数成员中, this 相当于结构类型的一个 ref 参数。两种情形之下, this 都表示一个变量, 可以通过对 this 赋值或把它作为一个 ref 或 out 参数传递来修调用改函数成员的整个结构。

11.3.7 域初始化 (Field Initializers)

如 11.3.4 节所述结构的默认值包括来自“清零”结构域的值。因此, 结构不许实例域声明含有变量初始化。下例是无效的。

```
struct Point
{
    public int x = 1; // Error, initializer not permitted
    public int y = 1; // Error, initializer not permitted
}
```

这个限制条件仅适用于实例域。结构的静态域可以含有变量初始化。

11.3.8 构造函数 (Constructors)

与类不同, 结构不能声明无参数构造函数。但每一结构固定拥有一个无参数的构造函数, 且该构造函数返回值总是由“清零”结构的域所产生。

结构构造函数不能有 base() 形式的构造函数初始化。

结构构造函数的 this 变量指的是结构类型的一个 out 参数, 与 out 参数一致, 在构造函数返回的每个位置, this 均需明确赋值 (参见 5.3 节)。

11.3.9 析构函数 (Destructors)

结构不能声明析构函数。

11.4 结构实例 (Struct Examples)

11.4.1 数据库整数类型 (Database Integer Type)

下述 DBInt 结构执行一整数类型, 该整数类型可表示整套 int 类型的值及指示一未知值的额外状态。具有这些特点的类型一般用于数据库中。

```
public struct DBInt
```

```
{
    // The Null member represents an unknown DBInt value.
    public static readonly DBInt Null = new DBInt();
    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.
    int value;
    bool defined;
    // Private constructor. Creates a DBInt with a known value.
    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }
    // The IsNull property is true if this DBInt represents an unknown value.
    public bool IsNull { get { return !defined; } }
    // The Value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.
    public int Value { get { return value; } }
    // Implicit conversion from int to DBInt.
    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }
    // Explicit conversion from DBInt to int. Throws an exception if the
    // given DBInt represents an unknown value.
    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }
    public static DBInt operator +(DBInt x) {
        return x;
    }
    public static DBInt operator -(DBInt x) {
        return x.defined? -x.value: Null;
    }
    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value + y.value: Null;
    }
    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value - y.value: Null;
    }
}
```

```

public static DBInt operator *(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value * y.value: Null;
}
public static DBInt operator /(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value / y.value: Null;
}
public static DBInt operator %(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value % y.value: Null;
}
public static DBBool operator ==(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value == y.value: DBBool.Null;
}
public static DBBool operator !=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value != y.value: DBBool.Null;
}
public static DBBool operator >(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value > y.value: DBBool.Null;
}
public static DBBool operator <(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value < y.value: DBBool.Null;
}
public static DBBool operator >=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value >= y.value: DBBool.Null;
}
public static DBBool operator <=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value <= y.value: DBBool.Null;
}
}
}

```

11.4.2 布尔型数据库类型 (Database Boolean Type)

下面的 DBBool 结构执行一个三值逻辑类型。该类型的可能值为 DBBool.True、DBBool.False、及 DBBool.Null, 这里 Null 表示一个未知值。在数据库中这种三值逻辑类型应用很广泛。

```

public struct DBBool
{
    // The three possible DBBool values.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
}

```

```
public static readonly DBBool True = new DBBool(1);
// Private field that stores -1, 0, 1 for False, Null, True.
sbyte value;
// Private constructor. The value parameter must be -1, 0, or 1.
DBBool(int value) {
    this.value = (sbyte)value;
}
// Properties to examine the value of a DBBool. Return true if this
// DBBool has the given value, false otherwise.
public bool IsNull { get { return value == 0; } }
public bool IsFalse { get { return value < 0; } }
public bool IsTrue { get { return value > 0; } }
// Implicit conversion from bool to DBBool. Maps true to DBBool.True and
// false to DBBool.False.
public static implicit operator DBBool(bool x) {
    return x? True: False;
}
// Explicit conversion from DBBool to bool. Throws an exception if the
// given DBBool is Null, otherwise returns true or false.
public static explicit operator bool(DBBool x) {
    if (x.value == 0) throw new InvalidOperationException();
    return x.value > 0;
}
// Equality operator. Returns Null if either operand is Null, otherwise
// returns True or False.
public static DBBool operator ==(DBBool x, DBBool y) {
    if (x.value == 0 || y.value == 0) return Null;
    return x.value == y.value? True: False;
}
// Inequality operator. Returns Null if either operand is Null, otherwise
// returns True or False.
public static DBBool operator !=(DBBool x, DBBool y) {
    if (x.value == 0 || y.value == 0) return Null;
    return x.value != y.value? True: False;
}
// Logical negation operator. Returns True if the operand is False, Null
// if the operand is Null, or False if the operand is True.
public static DBBool operator !(DBBool x) {
    return new DBBool(-x.value);
}
```



```
// Logical AND operator. Returns False if either operand is False,  
// otherwise Null if either operand is Null, otherwise True.  
public static DBBool operator &(DBBool x, DBBool y) {  
    return new DBBool(x.value < y.value? x.value: y.value);  
}  
  
// Logical OR operator. Returns True if either operand is True, otherwise,  
// Null if either operand is Null, otherwise False.  
public static DBBool operator |(DBBool x, DBBool y) {  
    return new DBBool(x.value > y.value? x.value: y.value);  
}  
  
// Definitely true operator. Returns true if the operand is True, false  
// otherwise.  
public static bool operator true(DBBool x) {  
    return x.value > 0;  
}  
  
// Definitely false operator. Returns true if the operand is False, false  
// otherwise.  
public static bool operator false(DBBool x) {  
    return x.value < 0;  
}  
  
}
```

第 12 章 数 组

一个数组就是一个含有很多变量的数据结构，其变量可通过对索引值来访问。数组中的变量，也称为数组元素，都是同一种类型。此类型称为数组的元素类型。

每个数组都有一个等级来限定引用每一数组元素的数目。数组的等级即数组的维。只具有一个等级的数组称为一维数组，具有多个等级的数组称为多维数组。

数组的每一维都有一个相关的长度，其值为大于或等于零的整数。维的长度不属于数组类型的一部分，但运行时它和数组类型一起产生。维的长度限定了其索引的有效等级：如某维的长度为 N ，其索引范围在 $0 \sim N-1$ 之间。数组中各维的长度即为该数组中元素的数目。若一个数组有一或多个维的长度为零，那么就说该数组是空的。

数组元素的类型可以为包括数组类型在内的任一类型。

12.1 数组类型 (Array Types)

数组类型写作 `non-array-type`，其后是 1 或多个 `rank-specifiers`：

```
array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    { dim-separatorsopt }

dim-separators:

    dim-separators
```

`non-array-type` 指的是本身并非 `array-type` 的任一 `type`。

`array-type` 中的左界 `rank-specifier` 给出数组类型的等级；`rank-specifier` 表明在 `rank-specifier` 中数组具有的等级及附加的“，”标识符的数目。

数组类型的元素类型由删除左界的 `rank-specifier` 产生。

- `T[R]` 形式的数组类型具有等级 R 及非数组类型 T 。
- `T[R][R1]...[RN]` 形式的数组类型具有等级 R 及元素类型 `T[R][R1]...[RN]`。

事实上，在最后的非数组元素类型之前，`rank-specifiers` 是由左向右读的。例如，`int[][,,][,]` 类型，即为一个 `int` 的二维数组的三维数组的一维数组。

具有一个等级的数组称为 `single-dimensional arrays`。具有多个等级的数组称为 `multi-`

dimensional arrays, 也可以称为二维、三维数组等等。

运行时, 数组类型的值可以是 null 或该数组类型实例的引用。

`System.Array` 类型是所有数组类型中的抽象基本类型。任一数组类型都可隐式转化为 `System.Array` (参见 6.1.4), `System.Array` 则可显式转化为任一数组类型 (参见 6.2.3 节)。

注意, `System.Array` 本身并非 array-type, 它是一个 class-type, 所有的 array-type 都由其产生。

运行时, `System.Array` 的类型值可以是 null 或任一数组实例的引用。

12.2 数组建立 (Array Creation)

Array-creation-expressions (参见 7.5.10.2)、域及包含一个 array-initializer (参见 12.6 节) 的局部变量声明都可以建立数组实例。

数组实例一经建立, 各维的等级及长度也就确定了, 并伴随该实例的始终。也就是说, 对于一个已经存在的数组实例来说, 其等级或维都不可改变。

由 array-creation-expression 建立的数组实例总是数组类型。`System.Array` 是抽象的, 无法用具体例子说明。

由 array-creation-expression 建立的数组元素总被初始化为其默认值 (参见 5.2 节)。

12.3 数组成员访问 (Array Element Access)

element-access 表达式 (参见 7.5.6.1 节) 可以访问数组成员, 形式为 `A[I1,I2,...,IN]`, 其中 `A` 是数组类型表达式, 各个 `IX` 是 `int`, `unit`, `long`, `ulong` 等类型或可隐式转化为这些类型的表达式。数组成员访问的结果是一个变量, 即由索引选定的数组元素。

使用 `foreach` 语句, 数组元素可以枚举 (参见 8.8.4 节)。

12.4 数组成员 (Array Members)

每一数组类型继承 `System.Array` 类型声明的成员。用 `foreach` 语句可以枚举数组成员。

12.5 数组方差 (Array Covariance)

对于任意两个 reference-types `A` 和 `B` 来说, 若从 `A` 到 `B` 存在一隐式引用转换 (参见 6.1.4 节) 或显式引用转换 (参见 6.2.3 节), 则从数组类型 `A[R]` 到数组类型 `B[R]` 也存在同样的引用转换。其中 `R` 是一给定的 rank-specifier (对于两个数组类型是一样的)。此关系称为 Array covariance。假设从 `B` 到 `A` 存在一隐式引用转换, 则数组方差的意思就是数组类型 `A[R]` 的值可能确实是对数组类型 `B[R]` 实例的一个引用。

由于数组方差的原因, 引用类型数组元素的赋值包括一个运行期检查, 以保证所赋值的类型合理 (参见 7.13.1 节)。例如:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
```

```

        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}

```

在 Fill 方差中 array[i] 的赋值固定含有一个运行期检查, 以保证 Value 引用的对象是 null 或者是一与该数组元素相容的类型的实例。在 Main 中, 前两次对 Fill 的调用成功了, 但是在第三次调用时, 执行第一次 array[i] 的赋值时 ArrayTypeMismatchException 即被抛出。这个异常发生的原因是封箱的 int 不能存储于 string 数组。

数组方差函数不能扩展到 value-type 的数组。例如, 不存在允许把 INT[] 当作一个 OBJECT[] 来对待的方差。

12.6 数组初始化函数 (Array Initializers)

数组初始化函数固定存在于域声明 (参见 10.4 节)、局部变量声明 (参见 8.5.1 节) 及数组建立表达式 (参见 7.5.10.2 节) 中:

```

array-initializer:
{ variable-initializer-listopt }
{ variable-initializer-list , }

variable-initializer-list:
variable-initializer
variable-initializer-list , variable-initializer

variable-initializer:
expression
array initializer

```

一个数组初始化函数包括一系列变量初始化, 包括在“{”和“}”之内, 以“,”隔开。每一变量初始化都是一个表达式, 或者, 对于多维数组来说, 是嵌套数组初始化函数。

使用数组初始化函数的上下文决定被初始化数组的类型。在数组建立表达式中, 数组类型在初始化函数之前。在域及变量声明中, 数组的类型即为声明的域及变量的类型。当数组初始化函数应用于域及变量声明中时, 如:

```
int[] a={0,2,4,6,8};
```

它是如下数组建立表达式的简写：

```
int[] a = new int[] {0, 2, 4, 6, 8}
```

对于一维数组来说，数组初始化函数必包括一系列表达式，它们是与该数组成员类型相匹配的赋值。以索引为零的成员开始，表达式以上升的次序初始化数组成员。数组初始化函数中表达式的数目限定了被建立的数组实例的长度。例如，上述数组初始化函数建立了长度为 5 的 `int[]` 实例，并初始化为下值：

```
a[0]=0; a[1]=2; a[2]=4; a[3]=6; a[4]=8;
```

对于多维数组而言数组初始化函数须有与该数组维数等量的嵌套水平最外界嵌套相当于最左界维。各维长度由该数组初始化函数中相应嵌套的成员数目决定对于每一个嵌套的数组初始化函数，其成员数目必须与同一水平其它的数组初始化函数相等。下例

```
int[,] b = {{0,1},{2,3},{4,5},{6,7},{8,9}};
```

建立了一个左界维长度为 5 的二维数组，其右界维长度为 2：

```
int[,] b = new int[5,2];
```

随后以下列值初始化该数组实例：

```
b[0, 0] = 0; b[0, 1] = 1; b[1, 0] = 2; b[1, 1] = 3; b[2, 0] = 4;
```

```
b[2, 1] = 5; b[3, 0] = 6; b[3, 1] = 7; b[4, 0] = 8; b[4, 1] = 9;
```

当数组建立表达式包括显式维长度及数组初始化函数时，其长度必须为常量表达式，且各个嵌套的成员数目必须与相应维的长度相匹配。例如：

```
int i = 3;
int[] x = new int[3] {0,1,2};           //OK
int[] y = new int[i] {0,1,2};           //Error, i not a constant
int[] z = new int[3] {0,1,2,3};         //Error, lenth/initializer mismatch
```

这里，`y` 的初始化函数是错误的，因为维长度表达式不是常量；`z` 的初始化函数也是错误的，因为初始化函数的元素数目和长度不一致。

第 13 章 接 口

一个接口定义一个契约。执行接口的类或结构必须遵守该契约。一个接口可能是从多个基本接口继承而来。类或结构可执行多个接口。

接口可包括方法、属性、事件及索引等。接口本身并不执行它定义的成员。接口只是指定由执行它的类或接口所提供的成员。

13.1 接口声明 (Interface declarations)

一个 interface-declaration 是声明一个接口类型的 type-declaration (参见 9.5 节)。

```
Interface-declaration:
    attributesopt      interface-modifiersopt      interface      identifier
    interface-baseopt interface-body      ;opt
```

一个 interface-declaration 包括一套可选择的 attributes (参见 17 章), 随之是可选择的 interface-modifiers (参见 13.1.1 节), 然后是关键字 interface 和命名该接口的 identifier, 可选择的 interface-base 说明 (参见 13.1.2 节), 之后是 interface-body, 最后, 是一个可选择的逗号。

13.1.1 接口修改函数 (Interface Modifiers)

一个 interface-modifier 包括可选择性的一系列接口修改函数:

```
interface-modifiers:
    interface-modifier
    interface-modifiers interface-modifier

interface-modifier:
    new
    public
    protected
    internal
    private
```

在一个接口声明中同一修改函数 重复出现是错误的。

new 修改函数 只允许出现在嵌套接口中。这说明在接口中隐藏着一个同名的继承成员, 如 10.2.2 节中所述。

修改函数 public, protected, internal 及 private 控制该接口的可访问性。在接口声明的上

下文中，可能只允许其中的几个修改函数（参见 3.5.1 节）。

13.1.2 基本接口（Base Interfaces）

一个接口可继承零到多个接口，称为接口的 **explicit base interfaces**。当一个接口有多个显式基本接口时，在接口声明中，接口标识符后是冒号和以逗号分开的基本接口标识符。

```
interface-base :
: interface-type-list
```

一个接口的显式基本接口至少与该接口本身具有相同的可访问性。例如，在 **public** 接口的 **interface-base** 中说明 **private** 或 **internal** 是不正确的。

接口直接或间接继承其本身是不正确的。

一个接口的 **base interfaces** 是显式基本接口及其基本接口。也就是说，这一系列的基本接口是此显式基本接口的完整的传递循环及其显式基本接口，等等。下例：

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

IcomboBox 的基本接口是 **Icontrol**，**IteatBox** 及 **IListBox**。

接口继承其基本接口的全部成员。即，上述 **IcomboBox** 接口继承 **SetText**、**SetItem**、及 **Paint** 的所有成员。

执行一接口的类或结构同时固定执行该接口的全部基本接口。

13.1.3 接口主体（Interface Body）

接口的 **interface -body** 定义该接口的成员。

```
interface -body:
{    interfacé-member-declarations opt    }
```

13.2 接口成员 (Interface Members)

接口的成员是从基本接口和接口本身声明的成员继承而来的。

```
interface-member-declarations:
    interface-member-declaration
    interface-member-declarations interface-member-declaration

interface-member-declaration:
    interface-method-declaration
    interface-property-declaration
    interface-event-declaration
    interface-indexer-declaration
```

一个接口声明可声明零或多个成员。接口成员必须是方法、属性、事件或索引。接口不能含有常量域操作符析构函数静态构造函数或类型。接口也不能含有任何类型的静态成员。

所有接口成员都固定具有公开的可访问性。接口成员中含有任何修改函数都是错误的。接口成员不能声明为 `abstract` , `public` , `protected` , `internal` , `private` , `virtual` , `override` 或 `static` 修改函数。

下例声明一个接口，它包括下列类型的一个成员：方法、属性、事件、及索引：

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

`interface-declaration` 建立一个新的声明空间（参见 3.3 节），由 `interface-declaration` 含有的 `interface-member-declarations` 在该声明空间中则立即建立新的成员。下列规则适用于 `interface-member-declarations`：

- 方法的名字必须与所有属性及该接口声明事件的名字不同。方法的签名(参见 3.6 节)必须与该接口声明的其它方法的签名不同。
- 属性及事件的名字必须与同一接口声明的其它成员的名字不同。
- 索引的签名必须与该接口声明的其它索引的签名不同。

特别地，接口的继承成员不是该接口声明空间的一部分。因此，接口可以声明与继承成员具有相同名字或签名的成员。此时，此派生接口成员将隐藏基本接口成员。隐藏继承成员不是错误，但会使编辑器发出警告。要消除此警告，派生接口成员的声明必须含有一个 `new` 修改函数，以表明此派生成员将隐藏基本成员。详情可参见 3.7.1.2 节。

若含有于声明中的 `new` 修改函数未隐藏继承成员，也将有警告显示。此警告可以通过去

掉 new 修改函数消除。

13.2.1 接口方法 (Interface Methods)

接口方法用 interface-method-declarations 声明:

```
interface-method-declaration:
    attributesopt newopt return-type identifier
    ( formal-parameter-listopt ) ;

interface-accessors:
    attributesopt get ;
    attributesopt set ;
    attributesopt get ; attributesopt set ;
    attributesopt set ; attributesopt get ;
```

接口属性声明中的 attribute, type 及 identifier 与类中属性声明中的意思相同。接口方法声明不能说明方法主体, 因此, 其声明总是以一个分号结束。

13.2.3 接口事件 (Interface Events)

接口事件用 interface-event-declaration 声明:

```
interface-event-declaration:
    attributes opt new opt event type identifier ;
```

接口事件声明中的 attribute, type 及 identifier 与类事件声明(参见 10.7 节)中的意思是相同的。除访问函数主体必须是一个分号外, 接口属性声明中的访问函数与类属性声明中的一致。因此, 访问函数的作用只是说明其属性为读写、只读或只写。

13.2.4 接口索引 (Interface Indexers)

接口索引用 interface-indexer-declaration 声明:

```
interface-indexer-declaration:
    attributes opt new opt type this [formal-parameter-list]
    { interface-accessors }
```

接口索引声明中的 attribute, type 及 formal-parameter-list 与类索引声明(参见 10.8 节)中的意思相同。

除访问函数主体必须为一个分号外, 接口索引声明中的访问函数与类索引声明(参见 10.8

节)中的一致。因此,访问函数的作用只是表明该索引为 读写、只读、或只写。

13.2.5 接口成员访问 (Interface Member Access)

接口成员通过成员访问 (参见 7.5.4 节) 及索引访问 (参见 7.5.6.2 节) 被访问, 表达式形式为 `I.M` 和 `I[A]`。其中, `I` 是一接口类型实例, `M` 是该接口类型的方法、属性、或事件; `A` 是索引函数自变量表。

对于严格单一继承的接口 (指继承链中每一接口有零或一个直属基本接口), 成员查找 (参见 7.3 节)、方法引用 (参见 7.5.5.1 节)、及索引访问 (参见 7.5.6.2 节) 的规则与类和结构的完全一样: 数目较多的派生成员隐藏具有相同名字或签名的较少派生成员。然而, 对于多个继承接口而言, 两个或更多的不相关基本接口声明具有相同名字或签名的成员时, 会产生歧义。下面是该情形实例。要解决歧义问题, 可在程序代码中插入显式 `cast`。下例:

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}

class C
{
    void Test(IListCounter x) {
        x.Count(1);           // Error, Count is ambiguous
        x.Count = 1;          // Error, Count is ambiguous
        ((IList)x).Count = 1;  // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1); // Ok, invokes ICounter.Count
    }
}
```

由于 `IListCounter` 中的 `Count` 成员查找 (参见 7.3 节) 不明确, 开始的两个语句在编译时出现错误。如上例中所示, 把 `x` 移到恰当的基本接口类型可解决此问题。此转移不占用运行时间——它们在编译时仅为较少派生类型的实例。

下例中对 `n.Add(1)` 的调用产生歧义。因为方法引用 (参见 7.5.5.1 节) 要求所有重载候选方法在同一类型中声明。而 `n.Add(1.0)` 的调用则是因为只有 `IDouble.Add` 是可以应用的。插入显式 `cast` 后, 只有一个候选方法, 因此就不会产生歧义。

```
interface IInteger
{
    void Add(int i);
}
```

```

    }
    interface IDouble
    {
        void Add(double d);
    }
    interface INumber: IInteger, IDouble {}
    class C
    {
        void Test(INumber n) {
            n.Add(1);          // Error, both Add methods are applicable
            n.Add(1.0);        // Ok, only IDouble.Add is applicable
            ((IInteger)n).Add(1); // Ok, only IInteger.Add is a candidate
            ((IDouble)n).Add(1);  // Ok, only IDouble.Add is a candidate
        }
    }
}

```

下例中 IBase.F 成员被 ILeft.F 成员隐藏:

```

interface IBase
{
    void F(int i);
}
interface ILeft: IBase
{
    new void F(int i);
}
interface IRight: IBase
{
    void G();
}
interface IDerived: ILeft, IRight {}
class A
{
    void Test(IDerived d) {
        d.F(1);          // Invokes ILeft.F
        ((IBase)d).F(1);  // Invokes IBase.F
        ((ILeft)d).F(1);  // Invokes ILeft.F
        ((IRight)d).F(1); // Invokes IBase.F
    }
}

```

因此, 虽然 `IBase.F` 看起来好像并未在 `IRight` 的访问路径中被隐藏, `d.F(1)` 的调用仍选择 `ILeft.F`。

在多个继承接口隐藏, 其规则非常简单、直观: 若一成员在一条访问路径中被隐藏, 则它在所有其它的访问路径中都将隐藏。因为 `IBase.F` 在从 `IDerived` 到 `ILeft` 及 `IBase` 的访问路径中被隐藏, 所以其成员也在从 `IDerived` 到 `IRight` 和 `IBase` 的访问路径中被隐藏。

13.3 全权接口成员名字 (Fully Qualified Interface Member Names)

有时接口成员需要用到 `fully qualified name`。接口成员的全权名字包括声明该成员的接口名字, 之后是一个句号, 最后是成员名。例如给定的声明:

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}
```

`Paint` 的全权名字是 `IControl.Paint`, `SetText` 的全权名字是 `ITextBox.SetText`。

注意, 成员的全权名字引用声明该成员的接口。因此, 上例中的 `Paint` 不是 `ITextBox.Paint`。

当接口为名字空间的一部分时, 接口成员的全权名字包括名字空间名。例如:

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

这里 `Clone` 的全权名字为 `System.ICloneable.Clone`。

13.4 接口执行 (Interface Implementations)

接口可以由类和结构执行。接口标识符被含有在类或结构的基类列表中以表明类或结构执行接口。

```
interface ICloneable
{
    object Clone();
}
```

```

}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}

    public int CompareTo(object other) {...}
}

```

执行接口的类或结构同时固定执行该接口的全部基本接口，即使该类或结构并未明显列出基类列表中的所有基本接口。

```

interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    public void Paint() {...}

    public void SetText(string text) {...}
}

```

这里，类 `TextBox` 同时执行 `IControl` 和 `ITextBox`。

13.4.1 显式接口成员执行 (Explicit Interface Member Implementations)

为了执行接口，类和结构可声明 `explicit interface member implementations`。显式接口成员执行是一个方法、属性、事件、或引用全权接口成员名字的索引声明。例如：

```

interface ICloneable
{
    object Clone();
}

```

```
interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}
}
```

这里 `ICloneable.Clone` 与 `IComparable.CompareTo` 是显式接口成员执行。

在方法调用、属性访问、或索引访问中通过其全权名字来访问一个接口成员执行。显式接口成员执行只可经接口实例访问，且只能被其成员名字引用。

显式接口成员执行不能含有访问修改函数，正如其不能含有 `abstract virtual`、`override`、及 `static` 修改函数一样。

显式接口成员执行具有与其它成员不同的可访问标志符。因为显式接口成员不能被其全权名字（方法调用及属性访问中）访问，它们是专一读取的。然而，由于它们可被接口实例访问，因此也是公共读取的。

显式接口成员有两个基本功能：

- 由于显式接口成员不可通过类及结构实例访问，因此，它允许类及结构的公共接口外的接口执行。这在类及结构执行一内部接口时尤其有用：该内部接口对类及结构的用户没有兴趣。
- 显式接口成员具相同签名的无歧义接口成员。缺少显式接口成员执行时，正如类及结构不能执行具有相同签名而返回类型却不同的任一接口成员，类及结构不能分别执行具有相同签名的接口成员并返回类型。

要保证显式接口成员执行有效，类及结构必须命名一个接口，其基类列表含有的成员的全权名字、类型、及参数类型与显式接口成员执行中的完全一致。因此，下面这个类中 `IComparable.CompareTo` 声明是无效的。因为，`IComparable` 未在 `Shape` 的基类列表中列出且不是 `ICloneable` 的基本接口。

```
class Shape : ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}
}
```

同样的，在声明：

```
class Shape : ICloneable
{
    object ICloneable.Clone() {...}
}
```

```
class Ellipse : Shape
{
    object ICloneable.Clone() {...}
}
```

中, Ellipse 中的 ICloneable.Clone 声明也无效, 因为 ICloneable 未在 Ellipse 中的基本类名中显式列出。

全权名字必须引用声明此成员的接口。因此, 以下声明中:

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

Paint 的显式接口成员执行必须写作 IControl.Paint。

13.4.2 接口映射 (Interface Mapping)

类和结构必须提供其基类列表中列出的全部接口成员的执行。在类及结构中, 接口成员执行定位的过程称为 interface mapping。

类或结构 C 的接口映射为列入 C 基类列表中的每一接口的每一个成员定位一个执行。特别的接口成员 I.M (I 为声明成员 M 的接口) 的执行由逐项检验每一类或结构 S 决定, 由 C 开始, 重复后面的基类 C, 直到一个匹配被定位:

- 如果 S 含有一个与 I 和 M 匹配的显式接口成员执行声明, 则此成员为 I.M 的执行。
- 否则, 如果 S 含有一个与 M 匹配的非静态公共成员声明, 则此成员为 I.M 的执行。

若执行不能为 C 的基类列表中的所有接口成员定位, 则错误出现。注意, 接口成员包括那些从基本接口继承而来的成员。

为便于接口映射, 在下列情况下类成员 A 与接口成员 B 匹配:

- A、B 都是方法, 且其名字、类型及标准参数列表一致。
- A、B 都是属性, 且其名字、类型一致, 有共同的访问函数。(若 A 不是显式接口成员执行可有附加访问函数。)
- A、B 都是事件, 且其名字、类型一致。

- A、B 都是索引，且其类型及标准参数列表一致。有共同的访问函数。（若 A 不是显式 接口成员执行可有附加访问函数。）

注意：接口映射算法的隐含规则是：

- 在确定执行接口成员的类或结构时，显式接口成员执行较类或结构的其它成员优先。
- 专用、保护及静态成员不参与接口映射。

在下例中 C 的 `ICloneable.Clone` 成员成为 `ICloneable` 中 `Clone` 的执行，因为显式接口成员执行有优先权：

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

如果一个类或结构执行二个或二个以上的接口，其成员具有相同的名字、类型及参数类型，那么就可以把这些接口成员映射到一个类或结构成员。例如：

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

这里，`IControl` 与 `IForm` 的 `Paint` 方法都被映射到 `Page` 中的 `Paint` 方法。也可以对两种方法分别实行显式接口成员执行。

如果一个类或结构执行一个含有隐藏成员的接口时，那么某些成员必须通过显式接口成员执行来执行。例如：

```
interface IBase
{
```



```

int p { get ; }
}
interface IDerived : IBase
{
new int p();
}

```

此接口的执行至少需要一个显式接口成员执行来执行，且采用下列形式中的一种：

```

class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}
class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}
class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}

```

当一个类执行具有相同基本接口的多个接口时，此基本接口只能有一个执行。例如：

```

interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
}

```

```
void ITextBox.SetText(string text) {...}  
void IListBox.SetItems(string[] items) {...}  
}
```

单独执行基类列表中命名的 `IControl`、由 `ITextBox` 继承的 `IControl` 及由 `IListBox` 继承的 `IControl` 都是不可能的。事实上，这些接口也没有独立的身份。`ITextBox`、`IListBox` 及 `IControl` 的执行是一样的。一般认为 `ComboBox` 只执行 `IControl`，`ITextBox` 及 `IListBox` 三个接口。

基本类成员参与接口映射。下例中 `class 1` 中的方法 `F` 用于 `interface 1` 中的 `class 2` 的执行：

```
interface Interface1  
{  
    void F();  
}  
class Class1  
{  
    public void F() {}  
    public void G() {}  
}  
class Class2: Class1, Interface1  
{  
    new public void G() {}  
}
```

13.4.3 接口执行继承 (Interface Implementation Inheritance)

一个类继承其基类提供的全部接口执行。

如果没有明显的 `re-implementing`，接口及派生的类不可能以任何形式改变从其基类继承的接口映射。例如在以下声明中 `TextBox` 中的 `Paint` 方法掩藏了 `Control` 中的 `Paint` 方法：

```
interface IControl  
{  
    void Paint();  
}  
class Control: IControl  
{  
    public void Paint() {...}  
}  
class TextBox: Control  
{  
    new public void Paint() {...}  
}
```

但是它并未将 `Control.Paint` 的映射改为 `IControl.Paint`。通过类实例及接口实例调用 `Paint` 将得到以下效果：

```
Control c = new Control() ;
TextBox t = new TextBox();
IControl ic = c ;
IControl it = t;

c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes Control.Paint();
```

然而，当类中的接口方法被映射到虚拟方法时，不可能用派生类来重载虚拟方法并改变接口的声明。例如，将上例的声明改写为下面的形式时：

```
interface IControl
{
    void Paint();
}

class control : Icontrol
{
    public virtual void Paint() {...}
}

class TextBox : Control
{
    public override void Paint() {...}
}
```

将会观察到以下效果：

```
Control c = new Control() ;
TextBox t = new TextBox();
IControl ic = c ;
IControl it = t;

c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes Control.Paint();
```

由于显式接口成员执行不能声明为虚拟，因此，不可能重载一个显式接口成员执行。然而，显式接口成员执行引用另一个方法却是有效的。该方法可以声明为虚拟，并允许派生类重载它。例如：

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

这里, Control 派生的类可以通过重载 PaintControl 方法来专门进行 IControl.Paint 的执行。

13.4.4 接口再执行 (Interface Re-Implementation)

如果基类列表中含有从一接口执行中继承的类, 则该类可 re-implement 那个接口。

接口再执行遵守其初执行的接口映射规则。继承的接口映射对于为接口再执行建立的接口映射无任何影响。例如在以下声明中 Control 映射 IControl.Paint 到 Control.Paint, 并不影响 MyControl 对它的再执行:

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

这里, MyControl 将 IControl.Paint 映射到 MyControl.Paint。

继承的公共成员声明及显式接口成员声明参与再执行接口的接口映射过程。例如:

```
interface IMethods
{
    void F();
}
```

```

    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}

```

这里，**Derived** 中的 **IMethod** 执行映射接口方法到 **Derived.F**，**Base.IMetgods.G**、**Derived.IMethods.H** 及 **Base.I**。

当一个类执行一个接口时，它同时也执行该接口的全部基本接口。一个接口的再执行也是其基本接口的隐式再执行。例如：

```

interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}

```

这里, IDerived 的再执行同时再执行了 IBase, 并将 Derived.F 映射到 D.F。

13.4.5 抽象类和接口 (Abstract Classes And Interfaces)

同非抽象类一样, 抽象类必须为列入该类的基类列表中的所有接口成员提供执行。但是, 抽象类允许映射接口方法到抽象方法。例如:

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

这里, IMethods 的执行映射 F 和 G 到抽象方法, 后者必须被从 C 派生的非抽象类重载。

注意: 显式接口成员执行不能是抽象的, 但可以调用抽象方法。例如:

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

这里, 从 C 派生的非抽象类将被要求重载 FF 和 GG, 这样就可以提供 IMethods 的实际执行过程。

第 14 章 枚 举

`Enum type` 是一种特殊的类型，它有已命名的常量。枚举声明可出现在类声明出现的地方。下例声明了一个名字为 `Color` 且带有成员 `Red`, `Green` 及 `Blue` 的枚举类型。

```
enum Color
{
    Red,
    Green,
    Blue
}
```

14.1 枚举声明 (Enum Declarations)

枚举声明声明一个新的枚举类型。枚举声明以关键字 `enum` 开始，定义该枚举的名字、可访问性、基础类型及成员等。

```
enum-declaration:
    attributesopt enum-modifiersopt enum identifier enum-baseopt
    enum-body ;opt

enum-base:
    : integral-type

enum-body:
    { enum-member-declarationsopt }
    { enum-member-declarations }
```

每个枚举类型都有称为该枚举类型的 `underlying type` 的相对应的整数类型。此基础类型必须说明该枚举定义的全部枚举值。枚举声明可以显式声明下列基础类型：`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` 及 `ulong` 等。注意 `char` 不能作为基础类型。未显式声明基础类型的枚举声明的基础类型为 `int`。下例中声明了一个基础类型为 `long` 的枚举：

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

开发者可以选择使用上例中的基础类型 `long` 来保证值的使用在 `long` 的范围内而非 `int` 的范围内，或为将来保留。

14.2 枚举修改函数 (Enum Modifiers)

`Enum modifier` 可以选择性的包括下列枚举修改函数：

```
enum-modifiers:  
enum-modifier  
enum-modifiers  enum-modifier  
  
enum-modifier:  
new  
public  
protected  
internal  
private
```

在同一个枚举声明中同一个修改函数不能重复出现。

枚举声明的修改函数与类声明的意思相同。注意，枚举声明不允许 `abstract` 及 `sealed` 修改函数。枚举不能是抽象的，也不能派生。

14.3 枚举成员 (Enum Members)

一枚举类型声明的主体定义零或多个枚举成员，它们是枚举类型带有名字的常量。枚举成员不能重名。

```
enum-member-declarations:  
enum-member-declaration  
enum-member-declarations  ,  enum-member-declaration  
  
enum-member-declaration:  
attributesopt  identifier  
attributesopt  identifier  =  constant-expression
```

每个枚举成员都有一个相关的常量值。该值类型为包含它的枚举的基础类型。每个枚举成员的常量值必须在该枚举基础类型范围之内。如下例即是错误的，因为 `-1`、`-2`、`-3` 不在 `unit` 的基础整数范围之内。

```
enum Color: unit  
{  
    Red=-1,  
    Green=-2,  
    Blue=-3
```


多个枚举成员可以共用一个相关值。例如：

```
enum Color
{
    Red,
    Green,
    Blue

    Max=Blue
}
```

表明一个枚举的两个成员- **Blue** 和 **Max**-具有同一个相关值。

枚举成员的相关值可以显式或隐式赋值。如果，枚举成员声明有一个 **constant-expression** 初始化函数，则该常量表达式的值（隐式转换为该枚举的基础类型）就是该枚举成员的相关值。如果枚举成员声明无初始化函数，则其相关值固定设置如下：

- 若该枚举成员为其枚举类型的第一个成员，则其相关值为零。
- 否则，枚举成员相关值为前一枚举成员的相关值加 1。这些值必须在基础类型所表示的值范围之内。

```
using System;
enum Color
{
    Red,
    Green = 10,
    Blue
}
class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }
    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);
            case Color.Green:
                return String.Format("Green = {0}", (int) c);
            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);
        }
    }
}
```

```

        default:
            return "Invalid color";
    }
}

```

上例打印输出枚举成员名字及其相关值如下：

```

Red=0
Blue=11
Green=10

```

原因如下：

- 枚举成员 Red 自动赋值为 0（因为它没有初始化函数且是第一个枚举成员）；
- 枚举成员 Green 显式赋值为 10；
- 枚举成员 Blue 自动赋值为较前一成员的值大 1 的值。

一枚举成员的相关值可以直接或间接地不使用其自身的值。除此循环限制外，不论其文本位置如何，枚举成员初始化函数可以任意引用其它枚举成员初始化函数。其它枚举成员的值被视为其基础类型的类型，因此，引用其它枚举成员时不需要转移。下例中是无效的，因为 A、B 的的声明是循环的：

```

enum Circular
{
    A=B,
    B
}

```

A 显式取决于 B；B 又隐式取决于 A。

枚举成员命名和分区的方式与类中域的方式非常相似。枚举成员的范围即为它包含的枚举类型的主体。在其范围内，枚举成员可以用其简单名字引用。与其它代码不同，枚举成员的名字必须给以枚举类型名字的权名。枚举成员没有任何声明的可访问性——含有可访问枚举类型的枚举成员可以被访问。

14.4 Enum 值和操作 (Enum Values And Operations)

每个枚举类型定义一个特殊类型；从 Enum 类型和整数类型以及两个枚举类型之间的转换都需要用显式枚举转换。枚举类型所取的一系列值不受枚举成员的限制。特别的，枚举基础类型的任何值都能被转移到该枚举类型，成为该枚举类型的一个特殊有效值。

枚举成员具有它所包含的枚举类型的类型（除非在其它枚举成员初始化函数内）相关值为 V 的枚举类型 E 所声明的枚举成员的值是 (E) V。下列操作符可用于枚举类型值：

```

=, !=, <, >, <=, >=, (~ (§ 7.9.5)), + (§ 7.7.4), - (§ 7.7.5), ^, &, | (§ 7.10.2), ~ (§ 7.6.4), ++, --

```

每个 Enum 类型自动从 System.Enum 类派生而来。因此，从该类中继承的方法及属性可用于枚举类型的值。

第 15 章 委托

委托保证 C++、Pascal、Modula 等其它语言给函数指针定址。与 C++ 函数指针不同，委托是完全面向对象的；与 C++ 指针成员函数不同，委托既压缩对象实例又压缩方法。

委托声明定义一个扩展 `System.Delegate` 类的类。一个委托实例压缩一个方法——一个可访问实体。对于实例方法来说，可访问实体包括一个实体和该实体的一个方法。对于静态方法来说，可访问实体只包括一个方法。只要有一个委托实例和一组合适的自变量，就可以用这些自变量来引用该委托。

委托的一个有趣且有用的特点就是它不知道也不关心它所引用的类的对象。只要方法的签名与委托的签名匹配，任何对象都可以。这使得委托非常适合“匿名”引用。

15.1 委托声明 (Delegate Declarations)

`delegate` -declaration 就是一种 `type-declaration` (参见 9.5 节)，它声明一个新的委托类型。

```
delegate-declaration:
    attributesopt delegate-modifiersopt delegate result-type
    identifier (formal-parameter-listopt )

delegate-modifiers:
    delegate-modifier
    delegate-modifiers delegate-modifier

delegate-modifier:
    new
    public
    protected
    internal
    private
```

在同一个委托声明中同一个修改函数不能重复出现。

`new` 修改函数只允许在其它类型中声明的委托中出现。如 10.2.2 节所述，委托隐藏与其同名的继承成员。修改函数 `public`, `protected`, `internal` 及 `private` 控制委托类型的可访问性。

有些修改函数在委托类型的上下文中不能使用 (参见 3.5.1 节)。

`formal-parameter-list` 识别委托的签名；`result-type` 说明委托的返回类型。委托的签名和返回类型必须与该委托类型压缩的任一方法的签名及返回类型完全匹配。C# 中的委托类型为名字一致，而不是结构一致。具有相同签名及返回类型的不同委托类型也被认为不同。

委托类型是从 `System.Delegate` 的类类型派生而来的。委托类型隐式 `sealed`：不能从委托类型派生任何类型。也不能从 `System.Delegate` 中派生非委托类型。注意，`System.Delegate` 本身并不是委托类型。所有的委托类型都是从它派生的。

对于委托实例和引用来说，C# 提供了专门的语法。除实例引用外，适用于类及类实例的

所有操作都适用于委托类及委托实例。但 `System.Delegate` 的成员可以一般成员访问语法来访问。

委托类型分为两类：`combinable` 与 `non-combinable`。一个可合并的委托类型须满足下列条件：

- 声明的委托返回类型为 `void`。
- 委托类型的参数不能声明为输出参数（参见 10.5.1.3 节）。

除非其中之一为 `null`，否则，如果合并（参见 7.7.4 节）两个不能合并的委托类型实例，那么运行时将出现错误。

15.2 委托实例 (Delegate Instantiation)

即使委托在很多方面与其它类一样，C# 仍为委托实例提供了专门的语法。`Delegate-creation-expression` (参见 7.5.10.3 节) 用以建立一个新的委托实例。

新建的委托实例包括：

- `Delegate-creation-expression` 中引用的静态方法，或者
- `Delegate-creation-expression` 中引用的目标对象（不可为 `null`）及实例方法，或者
- 其它的委托。

一旦实例化，委托实例总是引用同一目标对象和方法。

15.3 多 cast 委托 (Multi-Cast Delegates)

使用加法操作符（参见 7.7.4 节）可以将委托合并；使用减法操作符（参见 7.7.5 节）可以在一个委托中除去另一个委托。由合并两个或两个以上（非 `null`）的委托实例建立的委托实例称为 `Multicast` 委托实例。对于任一委托实例来说，该委托的 `invocation list` 定义为非多 `cast` 委托的次序表，在引用委托实例时它可被引用。具体来说：

- 对于非多 `cast` 委托来说，引用表包括委托实例本身。
- 对于由合并两个委托而建立的委托实例而言，引用表有合并多 `cast` 委托的加法操作符的操作对象的引用表产生。

15.4 委托引用 (Delegate Invocation)

C# 为引用委托提供了专门的语法。当一非多委托被引用时，同时以相同变量引用该委托所引用的方法，且返回的值与该方法返回的值相同。委托引用的详细情况可参见 7.5.5.2 节。

如果引用一委托时出现异常，且未在引用的方法内被捕获，寻找异常的子句将在引用该委托的方法内继续，就像该方法已直接引用了委托引用的方法一样。

多 `cast` 委托的引用在引用表中每一个引用之前。每一引用都通过同一组变量。如果该委托包括引用参数（参见 10.5.1.2 节），每个方法引用都引用同样的变量；引用表中的方法改变参数只能出现在引用表以后的方法中。

如果在引用一个多 `cast` 委托的过程中出现了异常，且未在引用的方法内被捕获，则寻找异常的捕获子句将在引用该委托的方法内继续，且引用表以后的方法将不再引用。

第 16 章 异 常

C#中的异常提供了一个有组织的、均一的、类型安全的处理系统水平和应用水平上错误的方法。关于异常的原理 C#与 C++非常相似，只有少数重要的差异：

- C#中的所有异常都由从 `System.Exception` 的类类型派生的实例来说明。C++中任何类型的值都可用来说明异常。
- C#中，最后的块（参见 8.10 节）可用于书写标准执行及在异常状况下执行的终止代码。在 C++中这种代码不能写，只能复制。
- C#中，诸如重载、零划分、空引用等系统水平的异常有一定定义的异常类且与系统水平出错状态对等。

16.1 异常原因 (Causes Of Exceptions)

异常可由两种方式抛出：

- `throw` 语句（参见 8.9.5 节）立即无条件的抛出异常。控制不立即跟随 `throw` 到达语句。
- 当操作不能被立即执行时，C#语句及表达式生成的某些异常状况将导致异常。例如，分母为零时整数除法操作符（参见 7.7.2 节）将抛出一个 `System.DivideByZeroException`。此情形下发生的各种异常可参见 16.4 节中的列表。

16.2 System.Exception 类 (The System.Exception Class)

`System.Exception` 类是所有异常的基本类型，此类有几个其它异常共有的属性：

- `Message` 是只读属性，包含一个只读的对异常原因的说明。
- `InnerException` 为只读属性，包括此异常的“内部异常”。若不是 `null`，表明当前异常是由于另一个异常而被抛出。导致该当前异常的异常可以在 `InnerException` 属性中获得。

这些属性的值可由 `System.Exception` 的构造函数来指定。

16.3 怎样处理异常 (How Exceptions Are Handled)

异常用 `try` 语句（参见 8.10 节）来处理。

异常发生时，同确定该异常的运行期类型一样，系统搜索最近的 `catch` 子句而不是处理异常。首先，为一个词法上相近的 `try` 语句搜索当前方法，然后依次考虑该 `try` 语句的相关 `catch` 子句。若这些方法失败了，将搜索称为 `try` 语句的方法及词法上相近的 `try` 语句的当前方法，该 `try` 语句含有一个调用到当前方法的指针。这种搜索一直继续到发现一个 `catch` 子句可以处理当前异常（通过命名一个同类的异常类、一个基类、或被抛异常的运行期类型）为止。不

命名异常类的 `catch` 子句可以处理任一异常。

匹配的 `catch` 子句一经发现，系统就把控制转到该 `catch` 子句的第一条语句。在 `catch` 子句执行之前，系统首先按次序执行较捕获此异常具有更多嵌套层的相关 `try` 语句的任一 `finally` 子句。

如果没有匹配的 `catch` 子句被发现，则下列其一将会发生：

- 若对匹配 `catch` 子句的搜索达到了一静态构造函数（参见 10.11 节）或静态域初始化函数，则 `System.TypeInitializationException` 将被转移到激发静态调用函数的调用点处。内部异常 `TypeInitializationException` 包括开始抛出的异常。
- 若对匹配 `catch` 子句的搜索到达了最初开始该线程或程序的代码，则该线程或程序的执行就被终止。

16.4 常用的异常类（Common Exception Classes）

表 16-1 中为 C# 操作符出现异常。

表 16-1 异常类

异常类	说明
<code>System.OutOfMemoryException</code>	定位存储（ <code>view</code> ）失败时
<code>System.StackOverflowException</code>	执行栈耗尽，过多的临近方法调用、出现很深或无限循环的典型征兆时
<code>System.NullReferenceException</code>	<code>null</code> 引用可导致需要引用对象的方式被使用时
<code>System.TypeInitializationException</code>	当静态构造函数抛出一个异常、且 <code>catch</code> 子句存在时
<code>System.InvalidCastException</code>	当基本类型或接口显式转换到一派生类型的运行期失败时
<code>System.ArrayTypeMismatchException</code>	由于存储元素的实际类型与数组实际类型不匹配而使存储一数组失败时
<code>System.IndexOutOfRangeException</code>	通过一小于零或超过该数组边界的索引来索引数组时
<code>System.MulticastNotSupportedException</code>	委托类型无 <code>void</code> 返回类型，致使合并两个非 <code>null</code> 委托失败时
<code>System.ArithmeticException</code>	对一个异常基类进行算术操作时，如 <code>DividedByZeroException</code> 及 <code>OverflowException</code>
<code>System.DividedByZeroException</code>	零除一整数值得时
<code>System.OverflowException</code>	算术操作符在一 <code>checked</code> 上下文中被重载时

第 17 章 属 性

多数 C# 语言都能使程序员指定程序中定义的实体的声明信息。例如，类中方法的可访问性通过用 `method-modifiers` `Public`, `protected`, `internal` 及 `private` 等修饰被指定。

C# 语言可使程序员发明新的声明信息，对各种程序实体指定声明信息，以及在运行环境中接受属性信息。例如，一个框架可以定义放置到程序元素(如类及方法)上的 `HelpAttribute` 属性，来提供从程序元素到文本的映射。

新类型的声明信息可由类属性（参见 17.1）声明来定义，这些类有位置和命名参数（参见 17.1.2 节），且可在运行时作为属性实例（参见 17.3 节）被重新得到。

17.1 类属性 (Attribute Classes)

`attribute class` 的声明定义了 `attribute` 的一个新类型，可以放在声明中。从抽象类 `System.Attribute` 派生的类，无论直接还是间接，都是一个类属性。根据惯例，类属性以后缀 `Attribute` 命名。使用属性时可含有或省略此后缀。

17.1.1 AttributeUsage 属性 (The AttributeUsage Attribute)

`AttributeUsage` 属性的作用是说明类属性的功能。

`AttributeUsage` 属性有一个已命名位置参数，它使得类属性指定可用的声明种类。下例定义了一个类属性，名为 `SimpleAttribute`，可置于 `class-declaration` 和 `interface-declaration` 中。

```
[AttributeUsage(AttributeTargets.Class| AttributeTargets.Interface)]

public class SimpleAttribute : System.Attribute

{ }
```

下例给出了 `Simple` 属性的几个用法。

```
[Simple] class Class1 { ... }

[Simple] interface Interface1 { ... }
```

该属性由名为 `SimpleAttribute` 的类定义，但使用时可省去后缀 `Attribute`，因此其名字简化为 `Simple`。上例和下例在语义上等同：

```
[SimpleAttribute] class Class1 { ... }

[SimpleAttribute] interface interface1 { ... }
```

`AttributeUsage` 属性有 `AllowMultiple` 参数，来决定该指定的属性对于一给定的实体可否

多次说明。若一属性的 `AllowMultiple` 为真，则它表示一个 `Multi-use attribute class`，对于一个实体可指定多次。若为假，则它表示一个 `single-use attribute class`，对于一个实体最多可指定一次。

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: System.Attribute {
    public AuthorAttribute(string value);

    public string Value { get {...} }
}
```

该例定义了一个名为 `AuthorAttribute` 的多用类属性。下例给出了一个两次使用 `Author` 属性的类声明。

```
[Author("Brian kernighan"), Author("Dennis Ritchie")]
class Class1 {...}
```

`AttributeUsage` 属性有一个 `Inherited` 命名的参数，当指定在一个基类中时，它说明该属性是否也被从其基类派生的类继承。如果 `Inherited` 参数未被说明，则使用 `false` 缺省值。

17.1.2 位置和命名参数 (Positional And Named Parameters)

类属性可有 `positional parameters` 和 `named parameters`。类属性的每个公共构造函数都为其定义一系列有效的位置参数。类属性的每个非静态公用读写域及属性都为其定义一个命名参数。

```
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: System.Attribute
{
    public HelpAttribute(string url) { // url is a positional parameter
        ...
    }

    public string Topic { // Topic is a named parameter
        get {...}
        set {...}
    }

    public string Url { get {...} }
}
```

该例定义了一个名为 `Attribute` 的类属性，它有一个位置参数 (`string url`) 和一个命名参数 (`string Topic`)。只读的 `url` 属性不定义命名参数。虽然它是非静态公用的，但由于它是只读的，所以不定义命名参数。

下例是类属性的几个用法:

```
[HelpAttribute("http://www.mycompany.com/.../Class1.htm")]
class Class1 {
;

[HelpAttribute("http://www.mycompany.com/.../Misc.htm", Topic ="Class2");]
class Class2 {
}
```

17.1.3 属性参数类型 (Attribute Parameter Types)

依据惯例, 类属性的位置和命名参数的类型被限定在 attribute parameter types 之内。下面的类型就是属性类型:

- 下列类型之一: bool、byte、char、double、float、int、long、short、string。
- object 类型。
- System.Type 类型。
- 可公开访问的枚举类型或其嵌套的类型 (如果有)。

17.2 属性说明 (Attribute Specification)

一个 attribute 就是一段附加的声明信息, 它说明一个声明。属性可定义在整个范围 (说明属性含有汇编或块), 且可为 type-declarations、class-member-declarations、interface-member-declarations、enum-member-declarations、property-accessor-declarations、event-accessor-declarations、formal-parameter 等声明定义。

属性定义在属性块内。属性块通常是在方括号中, 属性说明以逗号隔开。属性被说明的次序及在分区中的排列方式不重要。比如, 说明[A][B]、[B][A]、[A,B]、及[B,A]都是一样的。

```
attributes:
attribute-sections

attribute-sections:
attribute-section
attribute-sections attribute-section

attribute-section:
[ attribute-target-specifieropt attribute-list ]
[ attribute-target-specifieropt attribute-list ,]

attribute-target-specifier:
attribute-target :

attribute-target:
assembly
```

```
field
event
method
module
param
property
return
type

attribute-list:
attribute
attribute-list , attribute

attribute:
attribute-name attribute-argumentsopt

attribute-name:
reserved-attribute-name
type-name

attribute-arguments:
( positional-argument-list )
(positional-argument-list, named-argument-list )
( named-argument-list )

positional-argument-list:
positional-argument
positional-argument-list , positional-argument

positional-argument:
attribute-argument-expression

named-argument-list:
named-argument
named-argument-list , named-argument

named-argument:
identifier =attribute-argument-expression

attribute-argument-expression:

expression
```

一个属性包括一个 `attribute-name` 和一些可选择的位置和命名变量。位置变量（如果有）在命名变量之前。一个位置变量包括一个 `attribute-argument-expression`；一个命名变量包括一

个名字，然后是一个等号，最后是一个 `attribute-argument-expression`。

`attribute-name` 识别一保留的属性或属性。若 `attribute-name` 是 `type-name` 则此名字必须指的是一个类属性。否则，发生编译期错误。下例是错误的，因为它把非类属性的 `class1` 作为一类属性来使用。

```
class class1 {}

[Class1] class Class2 {} // Error
```

特定的上下文允许说明多个目标。如果含有 `attribute-target-specifier`，则程序可显式说明目标。除下面的上下文外，都可使用合理的缺省状态。因此，`attribute-target-specifiers` 经常被省略。这些潜在的有歧义的上下文是：

- 全局范围的属性说明适用于目标汇编或目标块。此上下文无缺省，因此 `attribute-target-specifier` 不可缺少。
- 委托声明的属性说明适用于委托声明本身或其返回值，无 `attribute-target-specifier` 时，该属性适用于委托声明。
- 方法声明的属性说明适用于方法声明本身或其返回值。无 `attribute-target-specifier` 时，该属性适用于方法声明。
- 操作符声明的属性说明适用于操作符本身及其返回值。没有 `attribute-target-specifier` 时，该属性适用于操作符声明。
- 非抽象事件声明的属性说明，缺省实际访问程序，可适用于事件声明本身或其自动关联的域。无 `attribute-target-specifier` 时，此属性适用于事件声明。
- 指定在特定的 `get` 访问函数中的属性说明适用于相关的方法或该方法的返回值。无 `attribute-target-specifier` 时，此属性适用于该方法。
- 指定在 `set` 访问函数中的属性说明适用于相关方法或该方法的单一参数。无 `attribute-target-specifier` 时，此属性适用于该方法。
- 事件的加减访问器的属性说明适用于相关方法或此方法的单一参数。无 `attribute-target-specifier` 时，此属性适用于该方法。

依据惯例，类属性以词缀 `Attribute` 命名。`Type-name` 形式的 `attribute-name` 可使用或省去词缀。`attribute-name` 和类属性名字最好匹配。如：

```
[AttributeUsage(AttributeTargets.All)]
public class X: System.Attribute
{
}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: System.Attribute
{
}

[X]           // refers to X
class Class1 {}

[XAttribute]  // refers to XAttribute
class Class2 {}
```

该例给出了两个名为 `X` 和 `XAttribute` 的类属性。属性 `[X]` 指类 `[X]`，属性 `[XAttribute]` 指类

属性[XAttribute]。如果类 X 的声明被移走，则两个属性都指的是类属性 XAttribute。

```
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: System.Attribute
{
    [X]                // refers to XAttribute
    class Class1 {}

    [XAttribute]        // refers to XAttribute
    class Class2 {}
}
```

同一实体中的单用类属性不能使用两次或两次以上。下例使用 HelpString 是错误的，因为它是个单用类属性，却在类声明中使用了多次。

```
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: System.Attribute
{
    string value;
    public HelpStringAttribute(string value) {
        this.value = value;
    }
    public string Value { get { ... } }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

若下列条件都符合，则表达式 E 就是一个 attribute-argument-expression:

- E 的类型为属性参数类型（参见 17.1.3 节）。
- 编译时，E 的值可为下列之一：
 - 一个常量值。
 - 一个 system.Type 对象。
 - attribute-argument-expression 的一个一维数组。

17.3 属性实例 (Attribute Instances)

一个 attribute instance 是在运行时表示一个属性的实例。属性以类属性、位置变量、及命名变量等定义。属性实例就是用位置和已命名变量初始化的类属性的实例。

如以下章节所述，属性实例的恢复包括编译期和运行期过程。

17.3.1 属性的编译 (Compilation Of An Attribute)

用类属性 T、positional-argument-list P 及 named-argument-list N 编译一个 attribute 时，包

括以下步骤:

- 编译一个 `New T(P)` 形式的 `object-creation-expression` 时的编译期步骤。这些步骤可能产生两个结果: 编译时出错或者确定一个运行时被调用 `T` 中的构造函数。称之为构造函数 `C`。
- 若上步中确定的构造函数无公开的可达访问性, 则编译时会出现错误。
- 对于 `N` 中每一个 `named-argument Arg`:
 - 让 `Name` 作为 `named-argument Arg` 的 `identifier`。
 - `Name` 必须在 `T` 中标识一个非静态读写公共域或者属性。如果 `T` 无那样的域或属性, 则编译时会发生错误。
- 为属性实例的运行期临时保留以下信息: 类属性 `T`, `T` 的构造函数 `C`, `positional-argument-list P` 及 `named-argument-list N`。

17.3.2 属性实例的运行期恢复 (Run-time Retrieval Of An Attribute Instance)

一个 `attribute` 的编译过程产生一个类属性 `T`, `T` 的构造函数 `C` `positional-argument-list P` 及 `named-argument-list N`。给定这些信息, 类属性实例在运行期的恢复, 有以下步骤:

- 使用编译时确定的构造函数 `C`, 执行 `T (P)` 形式的 `object-creation-expression` 时, 遵循运行期步骤。这些步骤可能有两个结果: 发生异常或者产生 `T` 的一个实例。称为实例 `O`。
- `N` 中的每一个 `named-argument Arg`, 按顺序依次为:
 - 令 `Name` 为 `named-argument Arg` 的 `identifier`。如果 `Name` 未在 `O` 上标识出一个非静态公共读写域或属性, 则将抛出异常。
 - 令求值 `Arg` 的 `attribute-argument-expression` 的结果为 `Value`。
 - 如果 `Name` 在 `O` 上标识了一个域, 则使其值为 `Value`。
 - 否则, `Name` 在 `O` 上标识一个 `Property`。赋其值为 `Value`。
 - 结果为 `O`, 由 `positional-argument-list P` 及 `named-argument-list N` 初始化的类属性 `T` 的一个实例。

17.4 保留属性 (Reserved Attribute)

少数属性在某些方面影响语言。这些属性包括:

- `System.AttributeUsageAttribute`, 其作用是说明类属性的使用方式。
- `System.ConditionalAttribute`, 其作用是定义条件方法。
- `System.ObsoleteAttribute`, 其作用是标记废除的成员。

17.4.1 AttributeUsage 属性 (The AttributeUsage Attribute)

`AttributeUsage` 属性的作用是说明类属性的使用方式。

被 `AttributeUsage` 属性修饰的类必须直接或间接的从 `System.Attribute` 派生而来。否则, 出现编译期错误。

```
[AttributeUsage(AttributeTargets.Class)]
public class AttributeUsageAttribute: System.Attribute
{
    public AttributeUsageAttribute(AttributeTargets validOn) {...}

    public virtual bool AllowMultiple { get {...} set {...} }

    public virtual bool Inherited { get {...} set {...} }

    public virtual AttributeTargets ValidOn { get {...} }
}

public enum AttributeTargets
{
    Assembly = 0x0001,
    Module = 0x0002,
    Class = 0x0004,
    Struct = 0x0008,
    Enum = 0x0010,
    Constructor = 0x0020,
    Method = 0x0040,
    Property = 0x0080,
    Field = 0x0100,
    Event = 0x0200,
    Interface = 0x0400,
    Parameter = 0x0800,
    Delegate = 0x1000,
    ReturnValue = 0x2000,

    All = Assembly | Module | Class | Struct | Enum | Constructor |
        Method | Property | Field | Event | Interface | Parameter | Delegate |
        ReturnValue,

    ClassMembers = Class | Struct | Enum | Constructor | Method |
        Property | Field | Event | Delegate | Interface,
}
```

17.4.2 Conditional 属性 (The Conditional Attribute)

Conditional 属性启动 conditional methods 的定义。Conditional 属性以前处理标识符的形式标示一种条件。是否调用条件方法取决于其符号是否在调用处被定义。如果符号被定义，就调用方法；如未被定义，则省略方法调用。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class ConditionalAttribute: System.Attribute
{
    public ConditionalAttribute(string conditionalSymbol) {...}

    public string ConditionalSymbol { get {...} }
}
```

条件方法必须服从下列规则:

- 条件方法必须为 **class-declaration** 中的方法。如果 **conditional** 属性指定在接口方法上, 则出现错误。
- 条件方法必须有一返回类型 **void**。
- 条件方法不能以 **override** 标识符来标注。它可用 **virtual** 标识符标注。重载方法是隐式附加条件的, 不可再被显式标以 **Conditional** 属性。
- 条件方法不能是接口方法的执行。否则将出现编译期错误。

条件方法用在 **delegate-creation-expression** 中也会出现编译期错误。

```
#define DEBUG

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

上例声明 **class1.M** 为条件方法。**class2** 的 **Test** 调用此方法。由于前处理符号 **DEBUG** 已被定义, 若 **class2.test** 被调用, 它将调用符号 **DEBUG**; 若未被定义, **class2.test** 不会调用 **class1.M**。

值得注意的是, 是否调用条件方法由该调用处的前处理标识符决定。

```
// Begin class1.cs
class Class1
{
    [Conditional("DEBUG")]
```

```
        public static void F() {  
            Console.WriteLine("Executed Class1.F");  
        }  
    }  
// End class1.cs  
  
// Begin class2.cs  
#define DEBUG  
  
class Class2  
{  
    public static void G {  
        Class1.F();           // F is called  
    }  
}  
// End class2.cs  
  
// Begin class3.cs  
#undef DEBUG  
  
class Class3  
{  
    public static void H {  
        Class1.F(); // F is not called  
    }  
}  
// End class3.cs
```

上例中 class2 和 class3 调用条件方法 class1.F，这是有条件的，主要是基于 DEBUG 的有无。由于此符号定义在 class2 的上下文而未在 class3 中，对 class2 中 F 的调用实现了。而 class3 中 F 未被调用。

在继承链中条件方法的使用具有迷惑性。通过式子 base.M 的 base 对条件方法的调用服从一般的条件方法调用规则。

```
// Begin class1.cs  
class Class1  
{  
    [Conditional("DEBUG")]  
    public virtual void M() {  
        Console.WriteLine("Class1.M executed");  
    }  
}
```



```

    }
}
// End class1.cs

// Begin class2.cs
class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M(); // base.M is not called!
    }
}
// End class2.cs

// Begin class3.cs
#define DEBUG

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M(); // M is called
    }
} // End class3.cs

```

上例中的 class2 包括 M 在其基类中定义的一个调用。此调用被省掉了，因为该基本方法是符号 DEBUG 的存在为条件的，而 DEBUG 未定义。因此，此方法在控制台上只写作“class2.M executed”。明智的使用 pp-declarations 可解决此问题。

17.4.3 Obsolete 属性 (The Obsolete Attribute)

Obsolete 属性被用来标明一个程序成员不可再使用。

```

[AttributeUsage(AttributeTargets.All)]
public class ObsoleteAttribute: System.Attribute
{
    public ObsoleteAttribute(string message) {...}
    public ObsoleteAttribute(string message, bool error) {...}
    public string Message { get {...} }
    public bool IsError { get {...} }
}

```

附录 A 不安全代码

如前章节所述, C#语言的核心程序与 C 及 C++显著的不同之处, 在于它将指针作为一个数据类型来省略。C#提供了引用及由垃圾收集函数操纵的建立对象的能力。此种设计与其它的特点一起, 使 C#语言较 C 及 C++更为安全。在 C#语言的核心程序, 未初始化的变量、“悬挂”的指针及超出一数组的界限的索引表达式, 都不可能存在。因此, 所有困扰 C 及 C++的故障都可被排除。

虽然在实践上 C 及 C++的各个指针类型结构都相应的在 C#中有一引用类型, 而有些情况下须得访问指针才行。例如, 连接基本操作系统、访问存储映射的装置及执行限时算法时, 都须访问指针。为了满足这些需要, C#提供了书写 `unsafe-code` 的能力。

在不安全代码中可声明及操作指针, 在指针及整数类型之间执行转换、获取变量的地址等等。从某种意义上说, 书写不安全代码就如同在 C#程序中书写 C 代码一样。

事实上, 不安全代码对于开发者及使用者都是“安全”标志。不安全代码须用修改函数 `unsafe` 明确标明, 因此开发者不可随意使用它, 编译函数与执行函数同时工作可保证不安全代码不会在不可靠行。

A.1 不安全环境 (Unsafe Contexts)

C#的不安全标志只可在 `unsafe contexts` 中获得。不安全环境由在一类型或成员的声明中含有 `Unsafe` 修改函数说明。

- 类、结构、接口及委托的声明可含有 `Unsafe` 修改函数, 该类型声明 (包括该类、结构、接口及接口主体) 的整个文本范围被认为是一个不安全环境。
- 域、方法、属性、事件、索引、操作符、构造函数、析构函数、及静态构造函数也可含有 `Unsafe` 修改函数, 该成员声明的整个文本范围也被认为是一个不安全环境。

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

此例的结构声明中的 `unsafe` 修改函数使该结构声明的整个文本范围成为一个不安全环境。这样, 就可把 `left` 及 `right` 域声明为指针类型。上例也可书写为:

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
```

```

    public unsafe Node* Right;
}

```

这里，域声明中的 `unsafe` 修改函数致使那些声明被认为是不安全环境。

建立一个不安全环境以允许指针类型的使用，除此之外 `unsafe` 修改函数对类型及成员无任何影响。

```

public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}
public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}

```

此例中，A 的 F 方法中的 `unsafe` 修改函数只是使 F 的文本范围成为一个不安全环境，在这个不安全环境中可能使用语言的不安全标志。B 中 F 的重载不需要 `unsafe` 修改函数——除非 B 的 F 方法本身需要访问不安全标志。当指针类型是方法签名的一部分时，情况略有不同。

```

public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B: A
{
    public unsafe override void F(char* p) {...}
}

```

这里，由于 F 的签名包括一个指针类型，它只能被书写在不安全环境中。然而，不安全环境的特点是可使整个类不安全（如 A 例），或是在方法声明中含有 `unsafe` 修改函数（如 B 中）。

A.2 指针类型 (Pointer Types)

在不安全环境中，一个 type 可以是 pointer-type、value-type 或 reference-type。

```
type:
value-type
reference-type
pointer-type
```

pointer-type 书写作 unmanaged-type 或关键字 void，其后是一个符号 “*”。

```
pointer-type:
unmanaged-type *
void *
unmanaged-type:
type
```

在指针类型中，*之前的类型称为该指针类型的 reference-type。它表示指针类型的值所指变量类型。

与引用不同无用单元的收集程序并不沿着指针轨道，它不知道指针及其所指的数据。因此，指针不可指向引用或含有引用的结构，且该指针的引用类型必须是 unmanaged-type。

Unmanaged-type 是非引用类型的任一类型，但在任何层次的嵌套中不包括 reference-type 域。也就是说，一个 unmanaged-type 是下列之一：sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal 或 bool:

- 任一 enum-type;
- 任一 pointer-type;
- 任一用户定义的只含有 unmanaged-type 的域的 struct-type。

混淆指针与引用的直观规则是引用的引用可以含有指针，而指针的引用则不能含有引用。指针类型的一些例子见表 A-1。

表 A-1 指针类型的例子

例 子	说 明
byte*	Pointer to byte
char*	Pointer to char
int**	Pointer to pointer to int
int*[]	Single-dimensional array of pointers to int
void*	Pointer to unknown type

指针类型 T* 的值表示变量的 address。指针间接操作符*可访问此变量。例如，给定一 int* 类型的变量 P，表达式 *P 表示从含有 P 的地址中找到 int 变量。与对象引用相似，指针可以为 null。应用间接操作符到一个 null 指针可引起一个 NullReferenceException 被显示。

类型 void 表示一未知类型的指针。由于引用类型未知，间接操作不能用于 void* 类型的指针。然而，void* 类型的指针可指向任一其它指针类型（反之亦然）。

指针类型是一独立的类型种类。与引用类型及值类型不同，指针类型不继承 `object`，且在指针类型与 `object` 之间不存在转换。指针尤其不支持封箱和非封箱（参见 4.3 节）。然而，允许在不同的指针类型及指针类型与整类型之间存在转换。详情见 A.4 节。

在一不安全环境中，操作指针时有以下几种结构：

- 操作符*的作用是执行指针的间接操作（参见 A.5.1 节）。
- 操作符->的作用是通过一个指针访问结构成员（参见 A.5.2 节）。
- 操作符[]的作用是索引一个指针（参见 A.5.3 节）。
- 操作符&的作用是取得一个变量的地址（参见 A.5.4 节）。
- 操作符++和--的作用是增加和减少指针（参见 A.5.5 节）。
- 操作符+和-的作用是指针运算（参见 A.5.6 节）。
- 操作符==, !=, (<), (>), (==, 及 !=)的作用是比較指针（参见 A.5.7 节）。
- `stackalloc`的作用是从调用堆栈分配内存（参见 A.7 节）。
- `fixed`语句的作用是暂时固定一个变量以得到其地址（参见 A.6 节）。

A.3 固定和可移动变量 (Fixed And Moveable Variables)

`address-of`操作符和（参见 A.5.4 节）和 `fixed`语句（参见 A.6 节）将变量分为两类：`fixed variable` 和 `moveables`。

固定变量存在于不被垃圾收集函数所影响的存储单元中。固定变量的例子包括局部变量、值参数及由引用指针建立的变量。可移动变量则存在于重新分配的或被垃圾收集函数处理的存储单元中。可移动变量的例子包括对象中的域及数组成员。

操作符&（参见 A.5.4 节）允许无条件获得固定变量的地址。然而，由于可移动变量属于垃圾收集函数的重新配置及处理，因此其地址只可由 `fixed`语句（参见 A.6 节）获得，且其只在 `fixed`语句的持续期有效。

确切地说，固定变量有以下几种：

- 由 `simple-name`（参见 7.5.2 节）产生的变量，指的是局部变量或值参数。
- 由式子 `V.I` 的一个 `member-access`（参见 7.5.4 节）产生的变量，其中 `V` 是 `struct-type` 的一个固定变量。
- 由式子 `*P` 的 `pointer-indirection-expression`（参见 A.5.1 节）、式子 `P>Z` 的 `pointer-member-access`（参见 A.5.2 节）或式子 `P[E]` 的 `pointer-element-access`（参见 A.5.3 节）产生的变量。

所有其它变量都是可移动变量。

注意：静态域是可移动变量。`Ref` 和 `out` 参数也是可移动变量，即使该参数给定的自变量是固定参数。最后，引用指针时产生的变量总是被归为固定变量。

A.4 指针转换 (Pointer Conversions)

在不安全环境中，一系列可用的隐式和显式转换被扩展到含有指针类型，如此节所述。

隐式指针转换可发生在不安全环境中的很多情况下，包括函数成员调用（参见 7.4.2 节）、`cast`表达式（参见 7.6.8 节）和赋值（参见 7.13 节）。隐式指针转换是指：

- 从任一 `pointer-type` 到 `void*` 类型的转换。
- 从 `null` 类型到任一 `point-type` 类型的转换。

显式指针转换只发生在不安全环境的 `cast` 表达式中（参见 7.6.8 节）。显式指针转换是指：

- 从某一 `pointer-type` 到任何其它 `pointer-type` 的转换。
- 从 `int`, `uint`, `long`, `ulong` 到任何 `pointer-type` 的转换。
- 从任何 `pointer-type` 到 `int`, `uint`, `long`, `ulong` 的转换。

关于隐式和显式转换的详情可参阅 6.1 节和 6.2 节。

两个指针类型之间的转换不改变指针值。也就是说，从一个指针类型到另一指针类型的转换对指针给出的基本地址没有影响。

指针与整数之间的映射依靠执行。然而在线性地址空间的 32 及 64 位 CPU 体系结构上，指针与整数类型之间的转换就如同 `uint` 或 `ulong` 值与整数类型之间的转换一样。

A.5 表达式中的指针（Pointers In Expressions）

在不安全环境中，一个表达式可产生一指针类型，但在不安全环境之外，表达式却不能是一指针类型。确切地说，在不安全环境之外，任一 `simple-name`（参见 7.5.2 节）、`member-access`（参见 7.5.4 节）、`invocation-expression`（参见 7.5.5 节）或 `element-access`（参见 7.5.6 节）都不能是指针类型。

在不安全环境中，`primary-expression`（参见 7.5 节）和 `unary-expression`（参见 7.6 节）的结果允许下列附加构造函数：

```
primary-expression:
    ...
    pointer-member-access
    pointer-element-access
    sizeof-expression
    stackalloc-expression

unary-expression:
    ...
    pointer-indirection-expression
    addressof-expression
```

这些构造函数在下面的章节中仍将叙述。

A.5.1 指针迂回（Pointer Indirection）

一个 `pointer-indirection-expression` 包括星号（*）及其后的一个 `unary-expression`。

```
pointer-indirection-expression:
    * unary-expression
```

此一元操作符*表示 `pointer indirection`，它的作用是获得指针所指向的变量。对*P（P 为一指针类型 T*的表达式）求值的结果是 T 类型的一个变量。把一元操作符*应用到 `void*` 类型的表达式或非指针类型的表达式中都是错误的。

把一元操作符*应用于 null 指针的影响是执行定义。但不能确定此操作是否会显示 `NullReferenceException`。

为便于明确赋值分析，由对式子*P 的表达式求值而产生的变量就被认为是明确赋值了。（参见 5.3.1 节）

A.5.2 指针成员访问 (Pointer Member Access)

一个 `pointer-member-access` 包括一个 `primary-expression`，然后是一个“`->`”，然后是一个 `identifier`：

```
pointer-member-access:
    primary-expression -> identifier
```

在指针成员访问的式子 `P -> I` 中，P 须为非 `void*` 的一个指针类型表达式，且 I 须表示 P 指向的类型的可访问成员。

式子 `P -> I` 的指针成员访问的求值结果是 `(*P).I`。指针间接操作符 `(*)` 的描述可参阅 A.5.1 节。成员操作符 `(.)` 的描述见 7.5.4 节。

```
struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + ", " + y + " ";
    }
}

class Test
{
    unsafe static void Main() {
        Point point;
        Point* p = &point;
        p->x = 10;
        p->y = 20;
        Console.WriteLine(p->ToString());
    }
}
```

此例中操作符 `->` 的作用是访问域及通过指针调用一结构的方法。由于 `P -> I` 与 `(*P).I` 是一致的，所以 `Main` 方法也可书写作：

```
class Test
{
    unsafe static void Main() {
```

```

    Point point;
    Point* p = &point;
    (*p).x = 10;
    (*p).y = 20;
    Console.WriteLine((*p).ToString());
}
}

```

A.5.3 指针成员访问 (Pointer Element Access)

一个 pointer-element-access 包括一个 primary-expression 及其后的一个在 “[]” 内的表达式。

```

pointer-element-access:

primary-expression [ expression ]

```

在式子 P[E] 的指针成员访问中，P 须为一个非 void* 的指针类型的表达式，E 须为可隐式转化为 int、uint、long 及 ulong 类型的表达式。

对式子 P[E] 的指针成员访问的求值结果为 *(P+E)。指针间接操作符 (*) 的描述详见 A.5.1 节。关于指针附加操作符 (+) 的描述可参阅 A.5.6 节。

```

class Test
{
    unsafe static void Main() {
        char* p = stackalloc char[256];
        for (int i = 0; i < 256; i++) p[i] = (char)i;
    }
}

```

指针成员访问的作用是初始化 for 循环中的字符缓冲区。由于 P[E] 与 *(P+E) 一致，上例也可书写为：

```

class Test
{
    unsafe static void Main() {
        char* p = stackalloc char[256];
        for (int i = 0; i < 256; i++) *(p + i) = (char)i;
    }
}

```

如 C 和 C++，指针成员访问操作符不检查超出界限错误，且访问一个出界成员的影响也不确定。

A.5.4 Address-of 操作符 (The Address-Of Operator)

一个 address-of-expression 包括一个 & 符号，然后是一个 unary-expression。

```
addressof-expression:
    & unary-expression
```

给定一个表达式 E (T 类型但被归类为一个固定变量 (参见 A.3 节))，结构 &E 将计算由 E 给出的变量的地址。结果类型为 T*，是一个值。如果 E 未被归为一个变量或 E 表示一个可移动变量，则错误出现。在后一种情况下，固定语句 (参见 A.6 节) 的作用是在获得其地址之前临时性地“安置”此变量。

操作符 & 不需要对其自变量进行明确赋值。但 & 操作之后，认为被操作的变量在操作发生的执行路径中就被明确赋值了。在这种情况下，程序有责任确保变量的初始化正确。

```
unsafe class Test
{
    static void Main() {
        int i;
        int* p = &i;
        *p = 123;
        Console.WriteLine(i);
    }
}
```

此例中认为 i 在初始化 p 的 &i 操作之后即被明确赋值。*p 的赋值影响了 i 的初始化，但此初始化是程序员的责任，若赋值被移走，将无编译错误发生。

操作符 & 的明确赋值规则可避免局部变量的过多初始化。例如，许多外部的 API 使指针指向一个充满 API 的结构。这些 API 的访问将越过局部结构变量的地址，若无此规则，则会出现结构变量过多的初始化。

A.5.5 指针增加和减少 (Pointer Increment And Decrement)

在不安全环境中，操作符 ++ 和 -- (参见 7.5.9 节和 7.6.7 节) 可用于除 void* 外的所有类型的指针变量。这样，对每一个指针类型 T* 来说，下列操作符都被隐式定义了：

```
T* operator ++(T* x);
T* operator --(T* x);
```

这些操作符产生同样的结果：从指针自变量中加上或减去整数值 1。也就是说，对于 T* 类型的指针变量，操作符 ++ 对变量中得到的地址增加 sizeof(T)；而操作符 -- 将对变量中得到的地址减去 sizeof(T)。

如果指针增加或减少操作超出了 CPU 结构的地址范围，结果是以执行依赖的方式被截断，不会出现异常。

A.5.6 指针运算 (Pointer Arithmetic)

在一个不安全环境中，操作符 + 和 - (参见 7.7.4 和 7.7.5 节) 可被用于除 void* 之外的所

有指针类型的值。这样，下列操作符隐式定义为：

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);
T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);
T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);
long operator -(T* x, T* y);
```

给定一个指针类型 T^* 的 P 表达式及类型 $\text{int}, \text{uint}, \text{long}, \text{ulong}$ 的 N 表达式，表达式 $P+N$ 与 $N+P$ 计算类型 T^* （由 P 给出的地址增加 $N * \text{sizeof}(T)$ 而得到）的指针值。同样，表达式 $P-N$ 计算类型 T^* （由 P 给出的地址减去 $N * \text{sizeof}(T)$ 而得到）的指针值。

给定两个指针类型 T^* 的表达式 P 和 Q ，表达式 $P-Q$ 计算 P 和 Q 给出的地址差值，然后再除以 $\text{sizeof}(T)$ 。结果的类型总是 long 。执行时 $P-Q$ 的计算方式如下：

```
((long) (P) - (long) (Q)) / sizeof(T)
```

如果一个指针运算操作超出了基本的 CPU 结构的地址范围，则结果就以执行依赖的方式被截断，不会出现异常。

A.5.7 指针比较 (Pointer Comparison)

在不安全环境中，操作符 $\text{==}, \text{!=}, \text{<}, \text{>}, \text{<=}, \text{>=}$ （参见 7.9 节）可应用于指针类型的所有值。指针比较操作符是：

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

由于隐式转换存在于任一指针类型与 void^* 类型之间，使用这些操作符可比较任一指针类型的操作对象。比较操作符，比较这两个操作对象给出的地址，且把它们当作未赋值的整数来对待。

A.5.8 sizeof 操作符 (The Sizeof Operator)

sizeof 返回给定类型变量占用的字节数。定义到 sizeof 的自变量类型须为 unmanaged-type (参见 A.2 节)。

```
sizeof-expression:
    sizeof (unmanaged-type)
```

sizeof 操作符的结果是一 int 类型的值。对于某些预先定义的类型来说, sizeof 产生一个常量, 如表 A-2 所示。

表 A-2 sizeof 产生的结果

表 达 式	结 果
sizeof(byte)	1
sizeof(char)	1
sizeof(short)	2
sizeof(ushort)	2
sizeof(int)	4
sizeof(uint)	4
sizeof(long)	8
sizeof(ulong)	8
sizeof(char)	2
sizeof(float)	4
sizeof(double)	8
sizeof(hool)	1

对于其它类型来说, sizeof 的结果是执行依赖的且被归为一个类型, 而不是常量。

A.6 固定语句 (The Fixed Statement)

fixed 语句用来“固定”一个可移动的变量使其地址在整个语句中保持不变:

```
fixed-statement:
    fixed (pointer-type fixed-pointer-declarators) embedded-statement
fixed-pointer-declarators:
    fixed-pointer-declarator
fixed-pointer-declarators , fixed-pointer-declarator
fixed-pointer-declarator:
    identifier=&variable-reference
    identifier=expression
```

每个 fixed-pointer-declarator 声明一个给定的 pointer-type 的局部变量, 该局部变量由相应的 fixed-pointer-initializer 计算的地址初始化。Fixed 语句中声明的局部变量在该声明左边的任

何 `fixed-pointer-initializers` 中及 `fixed` 语句的 `embedded-statement` 中都是可访问的。`Fixed` 语句声明的局部变量被认为是只读的，且不可被赋值或作为 `ref` 或 `out` 参数传递。

`fixed-pointer-initializer` 有以下几种：

- 符号 “&”，然后是一个到未处理类型 `T` 的可移动变量（参见 A.3 节）的 `variable-reference`（参见 5.4 节），假定类型 `T*` 可隐式转换到 `fixed` 语句给出的指针类型。在这种情况下，初始化函数计算给定变量的地址，此变量保证在 `fixed` 语句中保持比变。
- 具有未处理类型 `T` 的成员的 `array-type` 表达式，假定类型 `T*` 可隐式转换到 `fixed` 语句给出的指针类型。在这种情况下，初始化函数计算该数组的第一个成员的地址，且整个数组保证在 `fixed` 语句中保持不变。若数组表达式是 `null`，则显示 `NullReferenceException`。
- `string` 类型的表达式，假定 `char*` 类型可隐式转换到 `fixed` 语句给出的指针类型。在这种情况下，初始化函数计算该字符串首字符的地址，且整字符串字符保证在 `fixed` 语句中保持不变。若此字符串表达式是 `null`，则将显示 `NullReferenceException`。

对于每一个由 `fixed-pointer-initializer` 计算的地址来说，`fixed` 语句确保在其中此地址引用的变量不再由垃圾收集函数重新定位或处理。例如，若 `fixed-pointer-initializer` 计算的地址引用了一个对象的域或一数组实例的成员，`fixed` 语句将保证在此语句中，所包含的对象实例不再重新定位或处理。

程序员有责任保证 `fixed` 语句建立的指针不影响其它语句的执行。例如，当 `fixed` 语句建立的指针通过外部的 `APIS` 时，程序员须确保 `APIS` 不保留这些指针的任何内存。

`Fixed` 语句通过制造说明垃圾收集函数的表格而被执行，该垃圾收集函数的对象在可执行代码区保持不变。因此，只要垃圾收集函数过程在 `fixed` 语句执行期间未真正发生，则与该语句相关的代价将会很小。然而，当垃圾收集函数真正发生时，固定对象将导致堆栈的分离（因为它们不可移动）。因此，对象的固定只在确实需要时才发生，且时间越短越好。

```
unsafe class Test
{
    static int x;
    int y;
    static void F(int* p) {
        *p = 1;
    }
    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        fixed (int* p = &x) F(p);
        fixed (int* p = &t.y) F(p);
        fixed (int* p = &a[0]) F(p);
        fixed (int* p = a) F(p);
    }
}
```

此例列举了 `fixed` 语句的几种使用方法。第一个语句固定并获得了静态域的地址，第二个语句固定并获得了实例域的地址，第三个语句固定并获得了数组成员的地址。由于这些变量都是可移动变量，因此，在各例中都不能使用规则的 `&` 操作符。

上例中第三、第四 `fixed` 语句产生的结果相同。通常，对于一个数组实例 `a` 来说，在一个 `fixed` 语句中指定说明 `&a[0]` 与简单地指定说明 `a` 是一样的。

在取得数组实例 `a` 的指针 `P` 的 `fixed` 语句中，指针的值（从 `P` 到 `P+a.length-1`）表示该数组中成员的地址。同样，变量（从 `P[0]` 到 `[a.length-1]`）表示实际的数组元素。

```
unsafe class Test
{
    static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        fixed (int* p = a) Fill(p, 100, -1);
    }
}
```

该例中 `fixed` 语句的作用是固定一个数组以使其地址被传到接受指针的方法上。

由固定字符串实例所产生的 `char*` 值总指向一个空终止字符串。在为字符串实例 `S` 取得指针 `P` 的 `fixed` 语句中，从 `P` 到 `P+S.Length-1` 的指针值表示字符串中字符的地址，指针值 `P+S.Length` 总指向一个空字符（字符值 `'\0'`）。

因为字符串不可改变，程序员有责任确保被指针引用到固定字符串的字符保持不变。

自动零终止字符串在调用期望“C-style”字符串的外部 API 时尤其方便。然而，注意，字符串实例可以包括空字符。若存在那样的空字符，则如果它被认为是一个零终止 `char*` 时，此字符串将会被截断。

A.7 堆栈分配 (Stack Allocation)

在不安全环境中，局部变量声明（参见 8.5.1 节）可包括一个堆栈分配初始化函数，来分配调用堆栈中的内存。

```
variable-initializer:
    expression
    array-initializer
    stackalloc-initializer

stackalloc-initializer:
    stackalloc unmanaged-type [ expression ]
```

一个堆栈分配初始化函数（式子 `stackalloc T[E]`）要求 `T` 为一个未处理类型（参见 A.2 节）且 `E` 是一个 `int` 类型的表达式。结构分配调用堆栈分配 `E*SIZEOF(T)` 字节并为新分配的块产生 `T*` 类型的指针。如果没有足够的内存分配给某一给定大小的块，则显示：

StackOverflowException。

用 `stackalloc` 无法显示已分配自由内存。当函数成员返回时，在函数成员执行时建立的堆栈分配内存块则被自动地舍弃。这与 C 及 C++ 中的 `alloca` 函数的作用是一致的。

```
class Test
{
    unsafe static string IntToString(int value) {
        char* buffer = stackalloc char[16];
        char* p = buffer + 16;
        int n = value >= 0? value: -value;
        do {
            *--p = (char)(n % 10 + '0');
            n /= 10;
        } while (n != 0);
        if (value < 0) *--p = '-';
        return new string(p, (int)(buffer + 16 - p));
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

此例中，初始化函数 `stackalloc` 的作用是在 `IntToString` 方法中定位堆栈上的 16 位字符缓冲区。当该方法返回时，此缓冲区将被自动舍弃。

A.8 动态内存分配 (Dynamic Memory Allocation)

除 `stackalloc` 操作符外，C# 还提供预先定义的结构来管理垃圾收集函数的内存。此服务由支持类库或直接由基础操作系统输入提供。比如，以下 `memory` 类表明了 API 窗口的堆栈函数如何从 C# 访问：

```
using System;
using System.Runtime.InteropServices;
public unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    static int ph = GetProcessHeap();
    // Private constructor to prevent instantiation.
    private Memory() {}
}
```

```
// Allocates a memory block of the given size. The allocated memory is
// automatically initialized to zero.

public static void* Alloc(int size) {
    void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// Copies count bytes from src to dst. The source and destination
// blocks are permitted to overlap.

public static void Copy(void* src, void* dst, int count) {
    byte* ps = (byte*)src;
    byte* pd = (byte*)dst;
    if (ps > pd) {
        for (; count != 0; count--) *pd++ = *ps++;
    }
    else if (ps < pd) {
        for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
    }
}

// Frees a memory block.

public static void Free(void* block) {
    if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
}

// Re-allocates a memory block. If the reallocation request is for a
// larger size, the additional region of memory is automatically
// initialized to zero.

public static void* ReAlloc(void* block, int size) {
    void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// Returns the size of a memory block.

public static int SizeOf(void* block) {
    int result = HeapSize(ph, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}

// Heap API flags
```

```
const int HEAP_ZERO_MEMORY = 0x00000008;

// Heap API functions
[DllImport("kernel32")]
static extern int GetProcessHeap();
[DllImport("kernel32")]
static extern void* HeapAlloc(int hHeap, int flags, int size);
[DllImport("kernel32")]
static extern bool HeapFree(int hHeap, int flags, void* block);
[DllImport("kernel32")]
static extern void* HeapReAlloc(int hHeap, int flags,
    void* block, int size);
[DllImport("kernel32")]
static extern int HeapSize(int hHeap, int flags, void* block);
}
```

使用 `memory` 类的例子:

```
class Test
{
    unsafe static void Main() {
        byte* buffer = (byte*)Memory.Alloc(256);
        for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
        byte[] array = new byte[256];
        fixed (byte* p = array) Memory.Copy(buffer, p, 256);
        Memory.Free(buffer);
        for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
    }
}
```

此例通过 `memory`、`alloc` 分配 256 个字节的内存，且用从 0 到 255 的值初始化该内存块。然后它又分配了一个 256 个成员字节的数组，并使用 `memory`、`copy` 将内存块的内容复制到字节数组。最后，应用 `memory`、`free` 释放内存块，且该字节数组的内容在复制后输出。

附录 B 互操作性

本章中所描述的属性的作用是建立与 com 程序互用的程序。

B.1 ComAliasName 属性 (The ComAlias Name Attribute)

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Parameter |
        AttributeTargets.ReturnValue)]
    public class ComAliasName: System.Attribute
    {
        public ComAliasNameAttribute(string value) {...}

        public string Value { get {...} }
    }
}
```

B.2 ComImport 属性 (The ComImport Attribute)

当应用于类时, ComImport 属性将该类标记为外部执行的 com 类。这样的类声明使 C# 名字的使用指向一个 com 类。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class ComImportAttribute: System.Attribute
    {
        public ComImportAttribute() {...}
    }
}
```

由 comimport 属性修饰的类须遵守以下规则:

- 须同时为 anid 属性所修饰, 该属性为输入的 com 类指定说明 clsid。一个类声明含有 comimport 属性却不含有 anid 属性时将产生编译错误。
- 不能含有任何成员。(自动地提供一个无参数的公共构造函数)。
- 不能派生来自非 object 的类。

```
using System.Runtime.InteropServices;
```

```
[ComImport, Guid("00020810-0000-0000-C000-000000000046")]
class Worksheet {}

class Test
{
    static void Main() {
        Worksheet w = new Worksheet(); // Creates an Excel worksheet
    }
}
```

此例将 worksheet 类声明为一个从 com (具有一个“00020810-0000-0000-0000-000000000046”的 CLSID) 引入的类。worksheet 实例的例示导致一个相应的 com 例示。

B.3 ComRegister Function 属性 (The ComRegister Function Attribute)

ComRegister Function 属性出现在方法中, 表明在 com 登陆过程中此方法应被调用。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class ComRegisterFunctionAttribute: System.Attribute
    {
        public ComRegisterFunctionAttribute() {...}
    }
}
```

B.4 ComSource Interfaces 属性 (The ComSource Interfaces Attribute)

ComSource Interfaces 属性的作用是在输入的类中列出源接口。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class ComSourceInterfacesAttribute: System.Attribute
    {
        public ComSourceInterfacesAttribute(string value) {...}

        public string Value { get {...} }
    }
}
```

}

B.5 ComUnregister Function 属性 (The ComUnregister FunctionAttribute)

ComUnregister Function 属性出现在方法中，表明在 com 中所使用的汇编未被读数时应访问该方法。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class ComUnregisterFunctionAttribute: System.Attribute
    {
        public ComUnregisterFunctionAttribute() {...}
    }
}
```

B.6 ComVisible 属性 (The ComVisible Attribute)

ComVisible 属性的作用是说明一类或接口在 com 中是否可见。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class |
        AttributeTargets.Interface |
        AttributeTargets.Method)]
    public class ComVisibleAttribute: System.Attribute
    {
        public ComVisibleAttribute(bool value) {...}

        public bool Value { get {...} }
    }
}
```

B.7 DispId 属性 (The DispId Attribute)

dispId 属性的作用是说明一个 OLE 自动过程 dispId。dispId 是一个整数值，与 dispinterface 中的成员一致。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Field |
        AttributeTargets.Property)]
    public class DispIdAttribute: System.Attribute
    {
        public DispIdAttribute(int value) {...}

        public int Value { get {...} }
    }
}
```

B.8 DllImport 属性 (The Dllimport Attribute)

DllImport 属性的作用是指定 dll 的位置, 此过程包括一个外部方法的执行。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class DllImportAttribute: System.Attribute
    {
        public DllImportAttribute(string dllName) {...}

        public CallingConvention CallingConvention;

        public CharSet CharSet;

        public string EntryPoint;

        public bool ExactSpelling;

        public bool PreserveSig;

        public bool SetLastError;

        public string Value { get {...} }
    }
}
```

DllImport 属性有以下特点:

- 只能出现在方法声明中。
- 只有一个位置参数: 即 dllname 参数, 它说明含有输入方法的 dll 的名字。
- 命名参数有五个:
 - CallingConvention 参数, 说明入口的调用规则。若无指定的 CallingConvention, 则使用默认的 CallingConvention.Winapi。

- CharSet 参数, 说明入口的字符设置。若无指定的 CharSet, 则使用默认的 CharSet.Auto。EntryPoint 参数, 给定 dll 中入口的名字。若无指定的 EntryPoint, 则使用此方法本身的名字。
 - ExactSpelling 参数, 表明 Entrypoint 是否必须与所示入口的拼写完全匹配。若无指定的 ExactSpelling, 则使用默认的 false。
 - Preservesig 参数, 说明方法的签名是否应保留或改换。当一个签名将要被改换时, 其目标是一个具有 HRESULT 返回值且返回值为 retval 的附加输出参数。若无指定的 Preservesig 值, 则使用默认的 true。
 - SetLastError 参数, 说明该方法是否保存 Win32 “最后的错误”。若无指定的 SetLastError, 则使用默认的 false。
- 它是一个单用类属性。
- 另外, 由 DllImport 属性修饰的方法必须有 extern 修改函数。

B.9 Fieldoffset 属性 (The Fieldoffset Attribute)

Fieldoffset 属性的作用是说明结构的域的格式。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Field)]
    public class FieldOffsetAttribute: System.Attribute
    {
        public FieldOffsetAttribute(int value) {...}

        public int Value { get {...} }
    }
}
```

Fieldoffset 属性不能出现在类或成员的域声明中。

B.10 Guid 属性 (The Guid Attribute)

Guid 属性的作用是为一个类或接口指定一个全局唯一的标识符 (Guid)。此信息对与 com 的互动尤其有用。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Assembly
                    | AttributeTargets.Class
                    | AttributeTargets.Interface
                    | AttributeTargets.Enum
                    | AttributeTargets.Delegate
                    | AttributeTargets.Struct)]
```

```
public class GuidAttribute: System.Attribute
{
    public GuidAttribute(string value) {...}
    public string Value { get {...} }
}
}
```

位置字符串自变量的格式在编译期被检验。指定一个句法上无效的 GUID 的字符串自变量是错误的。

B.11 HasDefaultInterface 属性 (The HasDefaultInterface Attribute)

若存在, HasDefaultInterface 说明一个有默认接口的类。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class)]
    public class HasDefaultInterfaceAttribute: System.Attribute
    {
        public HasDefaultInterfaceAttribute() {...}
    }
}
```

B.12 ImportedFromTypeLib 属性 (The ImportedFormTypeLib Attribute)

ImportedFormTypeLib 属性的作用是说明一个由 com 类型库输入的汇编。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Assembly)]
    public class ImportedFromTypeLib: System.Attribute
    {
        public ImportedFromTypeLib(string value) {...}
        public string Value { get {...} }
    }
}
```

B.13 In 和 Out 属性 (The In and Out Attribute)

In 和 Out 属性的作用是提供参数的信息整理。这些整理属性可以联合使用。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Parameter)]
    public class InAttribute: System.Attribute
    {
        public InAttribute() {...}
    }

    [AttributeUsage(AttributeTargets.Parameter)]
    public class OutAttribute: System.Attribute
    {
        public OutAttribute() {...}
    }
}
```

若一参数不被整理属性修饰，则它将基于其 parameter modifiers 被整理。若此参数无修改函数则整理为 (In)，若此参数的修改函数为 ref，则整理为 (In,out)，若此参数的修改函数为 Out，则整理为 (Out)。

注意，out 是一个关键字，而 Out 是一个属性。

```
class Class1
{
    void M([Out] out int i) {
        ...
    }
}
```

此例表明 out 是一个 parameter，而 Out 是一个 attribute。

B.14 IndexerName 属性 (The IndexerName Attribute)

有些系统用索引过的属性执行索引函数。一个索引函数如果没有 IndexerName，则使用默认的名字 Item。IndexerName 属性使开发者可重载此缺省并指定另一个名字。

```
class Class1
{
    void M([Out] out int i) {
        ...
    }
}
```

B.15 InterfaceType 属性 (The InterfaceType Attribute)

当出现在接口中时, `InterfaceType` 属性表明 `com` 中该接口的处理方式。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Interface)]
    public class InterfaceTypeAttribute: System.Attribute
    {
        public InterfaceTypeAttribute(ComInterfaceType value) {...}
        public ComInterfaceType Value { get {...} }
    }
}
```

B.16 MarshalAs 属性 (The MarshalAs Attribute)

`MarshalAs` 属性的作用是说明域、方法、参数的整理格式。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method |
        AttributeTargets.Parameter |
        AttributeTargets.Field)]
    public class MarshalAsAttribute: System.Attribute
    {
        public MarshalAsAttribute(UnmanagedType unmanagedType) {...}
        public UnmanagedType ArraySubType;
        public string MarshalCookie;
        public string MarshalType;
        public VarEnum SafeArraySubType;
        public int SizeConst;
        public short SizeParamIndex;
        public int SizeParamMultiplier;
    }
}
```

B.17 NoIDispatch 属性 (The NoIDispatch Attribute)

`NoIDispatch` 属性表明输出到 `com` 的类或接口应从 `Iunknown` 而不是 `IDispatch` 派生而来。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
```



```

public class NoIDispatchAttribute: System.Attribute
{
    public NoIDispatchAttribute() {...}
}

```

B.18 Preservesig 属性 (The Preservesig Attribute)

Preservesig 属性的作用是说明 HRESULT/retval 签名的转化过程，这一般发生在互操作性访问应被压缩时。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Property)]
    public class PreserveSigAttribute: System.Attribute
    {
        public PreserveSigAttribute(bool value) {...}

        public bool Value { get {...} }
    }
}

```

B.19 StructLayout 属性 (The StructLayout Attribute)

StructLayout 属性的作用是说明结构域的模式。

```

namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
    public class StructLayoutAttribute: System.Attribute
    {
        public StructLayoutAttribute(LayoutKind value) {...}

        public CharSet CharSet;

        public bool CheckFastMarshal;

        public int Pack;

        public LayoutKind Value { get {...} }
    }
}

```

若指定了 LayoutKind.Explicit，则该结构的每一个域都必须有 structoffset 属性。若未指定 LayoutKind.Explicit，则禁止使用 structoffset。

B.20 TypeLibFunc 属性 (The TypeLibFunc Attribute)

在与 com 互操作中, TypeLibFunc 属性的作用是说明 typelib 的特征。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class TypeLibFuncAttribute: System.Attribute
    {
        public TypeLibFuncAttribute(TypeLibFuncFlags value) {...}
        public TypeLibFuncFlags Value { get {...} }
    }
}
```

B.21 TypeLibType 属性 (The TypeLibType Attribute)

在与 com 互操作中, TypeLibType 属性的作用是说明 typelib 的特征。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
    public class TypeLibTypeAttribute: System.Attribute
    {
        public TypeLibTypeAttribute(TypeLibTypeFlags value) {...}
        public TypeLibTypeFlags Value { get {...} }
    }
}
```

B.22 TypeLibVar 属性 (The TypeLibVar Attribute)

在与 com 互操作中, TypeLibVar 属性的作用是说明 typelib 的特征。

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Field)]
    public class TypeLibVarAttribute: System.Attribute
    {
        public TypeLibVarAttribute(TypeLibVarFlags value) {...}
        public TypeLibVarFlags Value { get {...} }
    }
}
```

B.23 支持枚举 (Supporting Enums)

```
namespace System.Runtime.InteropServices
{
    public enum CallingConvention
    {
        Winapi = 1,
        Cdecl = 2,
        Stdcall = 3,
        Thiscall = 4,
        Fastcall = 5
    }

    public enum CharSet
    {
        None,
        Auto,
        Ansi,
        Unicode
    }

    public enum ComInterfaceType
    {
        InterfaceIsDual = 0,
        InterfaceIsUnknown = 1,
        InterfaceIsIDispatch = 2,
    }

    public enum LayoutKind
    {
        Sequential,
        Union,
        Explicit,
    }

    public enum TypeLibFuncFlags
    {
        FRestricted = 1,
        FSource = 2,
        FBindable = 4,
        FRequestEdit = 8,
        FDisplayBind = 16,
        FDefaultBind = 32,
```

```
        FHidden = 64,
        FUsesGetLastError = 128,
        FDefaultCollelem = 256,
        FUiDefault = 512,
        FNonBrowsable = 1024,
        FReplaceable = 2048,
        FImmediateBind = 4096
    }

    public enum TypeLibTypeFlags
    {
        FAppObject = 1,
        FCanCreate = 2,
        FLicensed = 4,
        FPreDeclId = 8,
        FHidden = 16,
        FControl = 32,
        FDual = 64,
        FNonExtensible = 128,
        FOleAutomation = 256,
        FRestricted = 512,
        FAggregatable = 1024,
        FReplaceable = 2048,
        FDispatchable = 4096,
        FReverseBind = 8192
    }

    public enum TypeLibVarFlags
    {
        FReadOnly = 1,
        FSource = 2,
        FBindable = 4,
        FRequestEdit = 8,
        FDisplayBind = 16,
        FDefaultBind = 32,
        FHidden = 64,
        FRestricted = 128,
        FDefaultCollelem = 256,
        FUiDefault = 512,
        FNonBrowsable = 1024,
        FReplaceable = 2048,
        FImmediateBind = 4096
    }
```

```
}  
  
public enum UnmanagedType  
{  
    Bool= 0x2,  
    I1= 0x3,  
    U1= 0x4,  
    I2= 0x5,  
    U2= 0x6,  
    I4= 0x7,  
    U4= 0x8,  
    I8= 0x9,  
    U8= 0xa,  
    R4= 0xb,  
    R8= 0xc,  
    BStr= 0x13,  
    LPStr= 0x14,  
    LPWStr = 0x15,  
    LPTStr = 0x16,  
    ByValTStr= 0x17,  
    Struct= 0x1b,  
    Interface= 0x1c,  
    SafeArray= 0x1d,  
    ByValArray= 0x1e,  
    SysInt= 0x1f,  
    SysUInt= 0x20,  
    VBByRefStr= 0x22,  
    AnsiBStr= 0x23,  
    TBStr= 0x24,  
    VariantBool= 0x25,  
    FunctionPtr= 0x26,  
    LPVoid = 0x27,  
    AsAny= 0x28,  
    RPrecise= 0x29,  
    LPArray= 0x2a,  
    LPStruct= 0x2b,  
    CustomMarshaler = 0x2c,  
}  
}
```

参 考 文 献

- 1 Unicode Consortium.The Unicode standard, Version3.0.Addison-Wesley, Reading, Massachusetts, 2000. ISBN 0-201-616335-5.
- 2 IEEE.IEEE standard for Binary Floating-Point Annthmetic.ANSI/IEEE standard 754-1985。
网址: <http://www.ieee.org>.
- 3 ISO/IEC.C⁺⁺.ANSI/ISO/IEC 14882:1998.

C#编程思想

嘉木工作室 编著

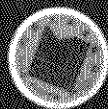


机械工业出版社



C# 编程思想

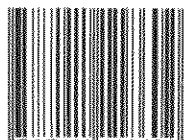
嘉木工作室 编著



● ISBN 7-111-11860-X/TP·2889

策划 边勇
封面设计 张静

ISBN 7-111-11860-X



9 787111 118602 >

定价: 36.00 元

地址: 北京市百万庄大街22号
联系电话: (010) 68326294

邮政编码: 100037

网址: <http://www.cmpbook.com>

E-mail: online@cmpbook.com

前 言

在过去的二十多年内，C 和 C++已经成为在软件开发中应用最广泛的开发语言。但是 C 和 C++的灵活性牺牲了它们的开发效率，如果和其他开发语言相比，相同功能的 C/C++软件通常需要更长的开发周期。正是由于 C/C++开发的复杂性和需要较长的开发周期，许多 C/C++开发人员都在寻找一种可以在功能和开发效率间提供更多平衡的开发语言。作为快速创建和集成 XML Web 服务和应用程序的单一综合工具，Visual Studio .NET 在改善操作的同时极大地提高了开发效率。其中的 Visual C# .NET（C#的发音为 C Sharp）是 Microsoft 推出的新型语言，它可提高 C 和 C++ 开发人员的效率。Visual C# .NET 对具有属性、方法、索引器、特性、版本和事件的组件提供一流的支持，并同时为 .NET 平台提供强有力的和有效的支持。

C#是一种先进的、面向对象的语言，通过 C#可以让开发人员快速建立大范围的基于 MS 网络平台的应用，并且提供大量的开发工具和服务，帮助开发人员开发基于计算和通信的各种应用。由于 C#是一种面向对象的开发语言，所以将 C#可以大范围的适用于高层商业应用和底层系统的开发。即使是通过简单的 C#构造也可以将各种组件方便地转变为基于 Web 的应用，并且能够通过 Internet 被各种系统或其他开发语言所开发的应用调用。

即使抛开上面所提到的优点，C#也可以为 C/C++开发人员提供快速的开发手段而不需要牺牲任何 C/C++语言的特点/优点。从继承角度来看，C#在更高层次上重新实现了 C/C++，熟悉 C/C++开发的人员可以很快的转变为 C#开发人员。C#的优点包括：

- 高的开发效率与安全性；
- 与 Web 开发相结合；
- 减小开发中的错误；
- 提供内置的版本支持来减少开发费用；
- 功能强，易于表现，灵活；
- 更好的结合商业应用中的流程与软件实现；
- 可扩展的协作能力；
- 允许有限制的使用指针。

总之，使用 C#能够利用方便快捷的 Microsoft 网络平台建立各种应用和建立能够在网络间相互调用的 Web 服务。从开发语言的角度来讲，C#可以更好地帮助开发人员避免错误，提高工作效率，并具有 C/C++的强大功能。

该书结构合理、内容翔实、循序渐进，适合各个层次的读者学习参考。由于时间仓促，本书编著过程中难免存在错误和纰漏，恳请广大读者批评指正。

该 C# 开发语言是包含在 Visual Studio.NET 框架中的最新应用程序开发语言。目前, 随着网络应用程序的广泛开发与应用需求, C# 已经成为开发基于计算和通信的最流行的语言。本书从基础入门, 结合 Web 开发的特点, 详细介绍 C# 开发语言的语法, 并针对这些语法提供了丰富的例程, 以充分发挥 C# 语言的开发优势。

本书适合从事网络程序设计的程序员、大中专院校相关专业师生和培训学校学生。

图书在版编目 (CIP) 数据

C# 编程思想/嘉木工作室编著. —北京: 机械工业

出版社, 2003.3

ISBN 7-111-11860-X

I. C... II. 嘉... III. C 语言—程序设计

IV. TP312

中国版本图书馆 CIP 数据核字 (2003) 第 025283 号

机械工业出版社 (北京市百万庄大街 22 号 邮政编码 100037)

策 划: 边 萌 责任编辑: 边 萌

封面设计: 张 静 责任印制: 闫 焱

北京交通印务实业公司印刷 · 新华书店北京发行所发行

2003 年 5 月第 1 版第 1 次印刷

787mm×1092mm×1/16 · 22 印张 · 554 千字

0 001—4000 册

定价: 36.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

本社购书热线电话 (010) 68993821、88379646

封面无防伪标均为盗版